6A.8 TOOLS FOR INTEGRATING DISTRIBUTED COMPUTING WITH INTERACTIVE VISUALIZATION IN MCIDAS-V

Raymond K. Garcia*, Bruce M. Flynn, Robert O. Knuteson, Thomas Whittaker, Thomas Rink, Thomas Achtor, Scott Mindock, Steven T. Dutcher, Maciej J. Smuga-Otto, Graeme D. Martin
University of Wisconsin Space Science & Engineering Center, Madison, Wisconsin

We present an overview and demonstration of open-source tools and technologies used to make large-scale computing connect readily to client visualization environments, bringing together multiple data sources to compose analyses in the McIDAS-V environment. McIDAS-V permits novel manipulations of atmospheric datasets distributed across the network using 3-D graphics and a highly literate data model implemented in Java. When coupled with plug-ins permitting it access to web services, cluster and grid computing can be made both easy-to-use and scriptable. Outputs can be sliced, subsetted and integrated into visualizations and further computations. This technology demonstration is intended to evolve into a toolkit and best practices for integrating heritage data processing applications with distributed computing and visualization.

1. INTRODUCTION

McIDAS-V is the latest in a line of visualization solutions for collating and analyzing meteorological data from diverse sources. McIDAS has, in many packages and generations, been supported by the University of Wisconsin SSEC for over 30 years. In recent years, the emergence of cluster and grid computing technologies has enabled scientists to access and process ever greater quantities of data, and various standard practices for grid and cluster computing have emerged for making these tasks easier for the end-user.

The present work shows a method for integrating best practices of cluster computing into the existing McIDASV visualization environment alpha release, by providing a plug-in interface for building and dispatching remote distributed computing tasks.

The principal kind of computation to be distributed involves "Job Farming," or taking a set of similar jobs that can each be executed independently of one another, and dispatching them to a cluster or grid for parallel execution. A proper mechanism for describing such jobs to the execution environments, as well as specifying the data sources needed for them is required for this to succeed. Distributed job farming traditionally involves tedious and error-prone construction of elaborate one-off scripts for a given algorithm. By constructing a set of practices embodied in a small framework and accompanying library, we hope to reduce the overhead typically required in taking a large experimental computation to a cluster, and to open up the experimenter's desktop or laptop to new possibilities for working with large quantities of atmospheric science data.

* Corresponding author address: Raymond K. Garcia, University of Wisconsin Space Science & Engineering Center, Madison, WI 53706; e-mail ray.garcia@ssec.wisc.edu.

2. REQUIREMENTS

From the point of view of computing resources, meeting our goal means making McIDAS V communicate effectively with compute clusters, grids and multiprocessor machines. An effective and flexible dispatching architecture is needed along with a McIDAS V plugin. Although the main focus of the work is on providing distributed computing capabilities to McIDAS V users, the software should also have command-line and scripting language API interfaces, both for testing purposes, and for later use in batching dispatch tasks. The McIDAS V plugin should ultimately provide a compelling graphical user interface (GUI), effectively communicating the capability of the computing systems it accesses. One of the great draws of an interactive visualization environment is its immediacy - no need to write batch files or other scripts.

In addition to describing jobs within the GUI, users need to be able to track the execution of longer-running jobs. A separate GUI element for monitoring the progress of requested work is therefore another requirement.

Our first iteration on this project was focused on using web technologies to create a data processing portal. This prior demonstration focused on general architecture and proof-of-concept over framework building or flexibility. The current iteration focuses on identifying tools and practices to help the developer bring new computations online, on scaling downward as well as upward, and on improving interactivity.

By "scaling downward", we address the requirement of McIDAS-V that it be able to run stand-alone on a single UNIX workstation such as a scientist's laptop. We add to this the requirement for running on machinery brought along on a field experiment, so that it can execute the same types of jobs as tasked for a distributed cluster and storage system, albeit on a smaller scale or at a slower rate. The McIDAS-V environment provides McIDAS-X functionality by way of a separable but easy-to-install McIDAS-X standalone server. We wanted to have a similar capability for parallel computation services.

By "scaling upward", we wish to be able to substitute alternate implementations of the subsystem interfaces that act as adapters to large clusters, grids, storage architectures, and remote archives. A multitude of systems exist for indexing, retrieving and accessing large data stores, including both open source and commercial solutions. Similarly, multiple queuing engines are used.

While ease-of use is a paramount requirement, the realities of distributed computing provide certain limits on what can be done in this regard. The use of clusters and grids often involves requirements for authentication and limitations on which clients can use what resources at any given time. While the current iteration of the project does not directly address these issues, its architecture must not preclude the eventual introduction of authentication and accounting to its capabilities.

Also, this system's usage practices must inherently encourage annotation of data with reproducibility (provenance) information. Results of computations are transitory while iteratively testing prototype algorithms. Making it easy to re-run a request with modified inputs or ancillary data, and knowing which output corresponded to which request, must be a consideration when producing and inspecting large amounts of data.

Finally, we were constrained by staffing and funding for this project. Executing these goals in an iterative fashion with a small development team of 2-3 has been both a benefit in that design goals and execution were largely precluded from committee debate, and a drawback in that this was one of many projects simultaneously vying for our attention.

Thus, our overall requirements for the system are to support parallel computing jobs involving large datasets, to provide a McIDAS V interface for interactive parallel job submission and output visualization with a straightforward UI, and to deliver the resulting software as a toolkit that can be deployed on systems ranging from standalone computers to clusters or grids. We need this system to allow developers to deploy algorithms rapidly and with good provenance practices, and we are using off-the-shelf, supported and proven technologies [Wallnau (2001)] to achieve the above goals in a reasonable time and effort.

3. ARCHITECTURE

In our original demonstration, Origami [Smuga-Otto (2007)], we made use of web application development frameworks, XML-RPC, and the UNIDATA Integrated Data Viewer (IDV) to show a large calculation being distributed across a cluster and the results stitched together in a single visualization, comparing observation against simulation.

While successful as a demonstration, more architectural work would be required in order to attain more flexibility while reducing the time to integrate the throng of algorithm prototypes which clamor for software developer time, large-scale parallelism and visualization capability.

As a result, we expanded on the three basic systems present in the original demonstration and focused on making the common integration cases easy, while not precluding future applications and extensions being more interactive and involving.

3.1 *Compute Facilities*

Logically, a single user request for a distributed processing run translates into a work-order. This work order consists of individual jobs, each job being able to run to completion independently of the others. These jobs can then be distributed, each job to a separate node, and upon completion their results can be gathered and presented to the user as the output to their work order. Work orders can often be decomposed into tasks along one or more dimensions: Spatial, temporal, channel, and ensemble. The specific types of decomposition available to an algorithm are generally known prior to integration of the algorithm with the computing system.

Division into spatially and temporally separate jobs is conceptually easiest: If an algorithm (such as a radiative transfer model) on a given region and time span has no external spatial or temporal dependencies, then a work order for running that algorithm on a large region can be split into separate jobs, each job running the algorithm on a spatial and temporal sub-domain. Some algorithms have complex spatial and temporal dependencies, but can subsect a work order into jobs by channel, band or profile level - a motivating example involves wind vector derivation that can be done independently by level.

Sometimes a work order can be subdivided not by spatial, temporal or other intrinsic divisions, but because the work requested involves running the same task with minor parameter tweaks, such as when evaluating and testing an algorithm against an error budget. One effort at the UW SSEC involves subjecting candidate algorithms to various combinations of bad or noisy input: These include combinations of radiometric noise, spatial mis-registration, missing bands, and other effects [Huang (2008)]. Testing an algorithm on all possible combinations of such effects is referred to as a test ensemble, and each individual test case can be spun into a separate job, enabling much faster execution of the entire ensemble.

Some work orders are decomposable in more than one way: An ensemble of tests may each be subject to further spatial and temporal independence. In a more complicated case, an end-to-end wind processing algorithm may first split inputs by space and time to derive profiles, then recombine and split again by levels to produce the final wind outputs.

Finally, work orders represent potentially complex series of tasks, involving combinations of FORTRAN, C++, MATLAB, Python, or any number of other computation environments. Not every work order type can be expected to be supported by every computing service. Some portions of a work order may have to execute in a dependency order - as is the case of spatially dividing a computation and then stitching the outcome together into an image.

Our first major subsystem, called the Work Order Dispatcher, addresses the above concerns. It provides profiles of work orders that the computing system supports, allows these profiles to be installed as part of a library of handler modules, and accepts work order requests from the clients. The dispatcher then uses the selected handler's planning function to produce a task break-down of the work order request. Lastly, it dispatches individual tasks to computing hardware, tracking their status and reporting back to the client. The dispatcher also provides the option to retain information on requests and tasks for future re-runs and provenance reporting.

3.2 *Storage Facilities*

We need a methodology for collecting the output of jobs for visualization and analysis. Typically, an algorithm will have any number of input and output files, including actual products, their metadata, and even log files. Moving files from place to place is an area of difficulty, when dealing with multiple generations of systems running different software versions of protocols and operating systems, and joined up with a variety of interconnects and authentication domains. Using local disk on individual compute nodes is most efficient, but also most problematic from when tracking the data down at a later date.

The simplest solution is to make data caching a service, on the understanding that pockets of accessible storage space are cheap to borrow for days or weeks at a time. In the simple single-machine case, this can wrap the existing "/tmp" present in some form on every machine. Tasks will then be responsible for submitting their output to the cache, ensuring the data is tagged by the work order that generated it. In our case, a storage area network (SAN) is our preferred landing point for data - this system holds dozens of terabytes of data and serves it through rsync and OPeNDAP protocols.

The cache server is capable of accepting data no matter the back end storage configuration, and based on its knowledge of available protocols on the storage system it represents, provides one or more URLs which can be used to access the content. We call the subsystem resident on the cache server the Data Cache Manager. It accepts file submissions from individual tasks and tags them with the relevant work order identifiers. It shares data with one or more access protocols represented by URLs that it generates. It allows clients to query for the list of results associated with a given work order identifier, and finally it cleans out unused and expired data periodically.

3.3 *Search Facilities*

Another challenging aspect of algorithm prototyping is identifying data sets to use for testing, assessing their quality and extracting subsets of practical size. This is made more complicated by a plethora of data file formats, varying extent of mark-up and metadata, various incompatible search and delivery mechanisms, and wide variance in provenance practices. Tackling all of the above at once is infeasible. As a pragmatic first cut, we had to satisfy ourselves with a simple geographical-temporal search on a well-defined data set, on the understanding that we would ultimately give similar treatment to potentially dozens of instrument (or instrument-like) datasets and databases.

We focused on product files from the IASI instrument. We built an IASI granule metadata database and data caching engine around this dataset for internal use in a matter of a few days' effort. This database focused on the most immediate features of interest, time and location, and included an extensible keywording capability to allow granules to be marked up based on identified features or relevance. Looking at other instrument databases, we saw the same trend of a handful of common fields with substantial mission or instrument-specific extensions.

The way the data is used varied. Some applications demand entire granules, others need only individual spectra within a search perimeter. Yet other views of the data include conjoining environment data records (EDR) with science data records (SDR) for a given test algorithm.

This and other experiences have led to several observations regarding instrument data search functionality:

- We do not yet have a sufficiently generalized schema for this domain to cover all uses of all instruments.
- Similarly, one definitive location for data has been found to be impractical.
- Search results are potentially complex, even hierarchical, depending on the application. They are likely to be useful as metadata files which can help direct computations and planning.
- We do not want to preclude the possibility of joining across multiple instrument databases or creating “pseudo-instrument” searches which index correspondence of multiple instruments.

The third major subsystem that handles the above functionality is called the Geo-Temporal Search. It accepts (at the least) spatial, temporal, and keyword-constrained searches, sends search result details to the Data Cache Manager which in turn can be used for work planning. The search service must of course provide descriptive output of the search results to the client, including URLs to the files and subset information found. Although the search as currently scoped only spans data residing in a single instrument catalog, the architecture should not preclude integration with search services that can perform "cross-database joins" for multiple instruments using multiple services

3.4 *The Client Interface*

All of the above functionality needs to be gathered together in a user-accessible interface. This interface component takes on several major use cases:

- *Interactive invocation*: Users will walk through a "data processing wizard" interface which allows them to identify appropriate input data and submit a processing request. They can then monitor progress of their work order and import the resulting products to a visualization environment.
- *Developer testing*: Algorithm integrators will need a robust set of tools for adding new algorithm handlers to a computing facility. Ultimately, this should progress to the point of having enough work archetypes and testing patterns that science staff will be able to do much of this on their own.
- *Standalone scripted automation*: Anything that users accomplish interactively can evolve into a batch script. The toolset should support easy access patterns for scheduled runs.
- *Embedded scripting automation (i.e. "UI buttons")*: Modern graphical development environments make possible the creation hybrid interfaces with the combination of a scripting system within a GUI.

Our overall system, incorporating the above components, is illustrated in Figure 1.

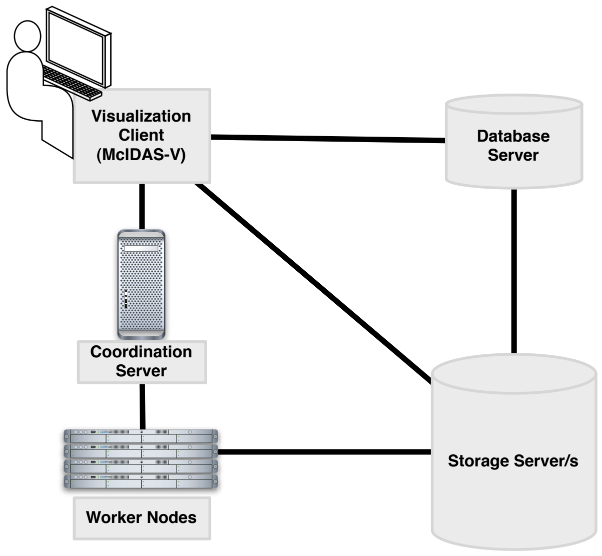

Figure 1. System Components.

4. TECHNOLOGY

A vital but often underestimated aspect of software architecture is the choice of underlying technologies. Once a project is settled on a particular platform or mix of platforms, such as Java, C#/.NET, CORBA, WSDL/SOAP, or UNIX, certain design opportunities open up, while others become complex or infeasible. The use of scripting languages such as Python eases some of these decisions, as these languages can act as glue between different platforms, but does not entirely remove the underlying platform strengths and weaknesses.

McIDAS V is principally a Java application, and as such leverages the considerable power and portability of the Java platform. Further capability and flexibility is added by the extensive use of Jython, an implementation of a subset of Python that works natively in the Java runtime environment.

Much of the original algorithm development and evaluation at the SSEC takes place in MATLAB, however use of this proprietary coding environment makes it hard to integrate with larger open standard based systems, except by means of batch scripting. Prototype FORTRAN and C++ algorithm implementations are often no more amenable to reworking, and present rigid interfaces that must be integrated into larger systems as delivered. Major re-factorings must often be deferred until the algorithm reaches a later stage in its development cycle.

For the purposes of this project, the NetCDF file format is the gold standard, allowing meaningful metadata to be embedded alongside raw data. The OPeNDAP server architecture both sources and provides NetCDF files to remote processes, making the OPeNDAP - NetCDF system very flexible in a distributed computing situation. Use of accepted metadata formats as specified in the UNIDATA "CF" standard and UDUNITS increase the longevity of a dataset. Data that use these metadata standards also automatically support unit conversion and implicit data mappings in VisAD / IDV / McIDAS V.

With the above platform challenges, we sought a maximally flexible set of technologies to glue together the various components of the target architecture. At first, SOAP (Simple Object Access Protocol) and WSDL (Web Service Description Language) were favored for interprocess communication, as they are based on open cross-platform standards, and are the current preferred solution for web based distributed applications. Soon however, we discovered that SOAP is anything but simple, and getting this mix of technologies to act as a communication channel exhibited a substantial barrier to productivity. Instead of pursuing the SOAP/WSDL route, we switched to CORBA as our communications channel.

CORBA was developed over a decade ago to solve a similar problem set to SOAP and WSDL, but before the XML and web-enabled technologies became ubiquitous. It had initially acquired a reputation for being unwieldy and poorly implemented, but years of incremental development have produced an eminently usable system for distributed computing in a largely language-agnostic fashion. While it is not considered cutting edge, it is a mature technology and fulfills the project needs directly, doing so with a minimum of custom configuration.

A downside to using CORBA for many is that it is not as well-suited as WSDL+SOAP for widely distributed, loosely coupled systems: XML serialization over the HTTP port is arguably superior (or at least, more often found in practice) for reaching through firewalls and across a wider range of languages. However, our experiments with SOAP and the overall project scope led us to CORBA as a highly practical and productive technology choice, well suited to our project scope.

```
struct CacheEntry {
  string uuid;                // of the work order that it belonged to
  string owner;               // contact point
  boolean is_search;          // if this is true, the cache entry is a search result
                              // and not data processing output
  double incept;              // creation time as second since UNIX epoch
  double expiry;              // time when this will be flushed, or NaN
  sequence<CacheLink> links;  // alternate access routes to this content
};

struct CacheRequest {
  string uuid;                // work order to associate the file with
  string filename;            // for the data stream to be stored as
  ...
};

interface DataCacheManager {
  // query routine for data consumers
  CacheEntrySeq resultsForWorkOrder( in string uuid );

  // reservation and cache filling routine for data providers
  // note that completing a single work order may result in several cache pickups
  CacheEntry requestCachePickup( in CacheRequest data );
};
```
Figure 2. Excerpt of DataCacheManager interface (CORBA IDL).

```
struct WorkOrderProfile {
  string uuid;                    // uuid of the profile - when functionality changes,
uuid changes
  string name;                    // name of the work order (title)
  string desc;                    // description of what it does, potentially lengthy
  string uri;                     // e.g. python:module.submodule#function
  sequence<FlagInfo> flags;       // boolean checkbox descriptions
  sequence<OptionInfo> options;   // pull-down options to select among
  sequence<FieldInfo> fields;     // fill-in text boxes for URLs et cetera
};

struct WorkOrderContent {
  string uuid;                    // uuid of the profile to run
  sequence<boolean> flags;        // flag values, in order corresponding to profile
  sequence<string> options;       // option values, in order corresponding to profile
  sequence<string> fields;        // field contents, in order corresponding to profile
  string output_ior;              // DataCacheManager instance IOR to receive output
                                  // to, empty if use-default
};

struct WorkOrderStatus {
  string uuid;                    // uuid of the work request this status reflects
  WorkState state;                // enum: queued, running, done, incomplete, aborted
  string desc;                    // description of the current status of the job
  string output_ior;              // IOR of DataCacheManager receiving the data
};

interface WorkOrderDispatcher {
  // provide list of profile structures describing flags, options, fields
  WorkOrderProfileSeq discoverWorkOrderProfiles();
  // submit a work order for processing and recover its initial status
  WorkOrderStatus submitWorkOrder( in WorkOrderContent wo_data );
  // query the most recent status of a submitted work order, given its UUID
  WorkOrderStatus queryWorkOrderStatus(in string woid);
};
```
Figure 3. Excerpt of WorkOrderDispatcher interface (CORBA IDL).

```
struct SearchForm {
  // time span to intersect
  TimePeriod span;
  // area to intersect
  sequence<Point> perimeter;
  // boolean expression to be evaluated on dataset keyword-value pairs as a filter
  string keyword_expr;
  // any additional search instructions which may be specific to this data type
  sequence<string> extra;
  // optional (empty string if not provided) - existing search result to start from
  // this would expand a previous result to "drill down" into smaller detail,
  // for instance instrument -> granule -> scan -> field -> spectrum
  string container_uuid;

  // optional - suggested datacachemanager to publish hit-lists to
  string output_ior;
};

interface GeoTemporalSearch {
  SearchResultSeq search( in SearchForm request );
};
```

Figure 3. Excerpt of GeoTemporalSearch interface (CORBA IDL).

5. IMPLEMENTATION

The platform architecture we settled on was a CORBA-IPC system written in Java and Python.

For Java, the *org.omg.corba* built-in library provided our client-side access, with stubs compiled from the three IDL interface descriptions: GeoTemporalSearch.idl, DataCacheManager.idl, and WorkOrderDispatcher.idl. For Python, the most straightforward route was to use *ORBit-python*, a mature CORBA object request broker. It is already found on many Linux systems as a middleware for the GNOME desktop. Marshaling of our relatively simple data structures and small number of calls was easy and obeys each programming language's idioms due to extensive work by the CORBA community on standard language bindings.

This freed us to work rapid prototyping on each service in a few hundred lines of Python. It also did not preclude future implementations being done in Java or even C/C++ as the need may arise. It has the added advantage of adhering to the free software principles adopted by the McIDAS-V development team.

We started with test implementations of the Work Order Dispatcher (WOD) and Data Cache Manager (DCM). These first cuts were intended to function as "sniffer" tools for interface testing, to allow us to refine our initial interfaces and to allow parallel development between our Java UI development and Python service development. The sniffer tools could then evolve into a command-line toolset.

Links between services and client are initialized using CORBA "IOR" strings which are uploaded to an accessible location at startup and provided in the form of a readable URL. This provides a stable reference location while not precluding future use of other more elaborate service advertising or naming systems, such as zeroconf.

Following that, we started into production versions of the three major services.

5.1 *The DataCacheManager*

The production DCM would have the capability to ship data through RSYNC to our SAN system, and mark up NetCDF and HDF files with their equivalent OPeNDAP URLs on that server. It is the simplest of the three services - its responsibilities are limited to keeping track of data files, returning URLs to them, and pushing them to a large disk pool for service by Apache, Hyrax (OPeNDAP), FTP and/or RSYNC. Its secondary function is to operate on a local machine as a standalone service and offer local file URLs.

The major calls for the DCM are shown in Figure 2.

5.2 *The WorkOrderDispatcher*

The WOD was the most complex of the three to implement from scratch. It has a library of dynamically loadable work order handler modules. Each module has a *.profile* structure which answers to a CORBA interface advertising the flags, options and fields that must be provided in order for a request to be dispatched. It also has a .plan() function which takes a submission and returns Task objects which must be scheduled. Finally, each work order module has one or more task functions which are ultimately executed in slave processes by way of a "glider" host script. This script is spawned on each slave CPU to execute a work order task(). In the case of an exception in the task, its hosting glider returns (through the shared task table database) an exception trace log appropriate for debugging.

The service can operate independently using a simple built-in task scheduler - this is used on standalone systems and multiprocessor boxes. If a queueing system is available (e.g. Sun Grid Engine), it is straightforward for it to submit to a pre-existing queue. An SQL database is used to track tasks and their completion - in the single-machine case an SQLITE database file is sufficient, while a Postgres or MySQL table is easily set up for cluster installs. Using a database server helps with maintaining visibility into the system as it operates and reduces the number of potential races when dealing with multiple processes.

The principal calls of the WorkOrderDispatcher are illustrated in Figure 3.

5.3 *The GeoTemporalSearch*

The GTS was the third to be implemented, essentially as an adapter to the previously-deployed database of selected IASI data. The initial implementation serializes geographic search results describing per-granule subsets as Python "pickle" files; future versions are likely to serialize as JSON or an XML format. These “hit-list” files are cached with the DCM and are used by work order handlers in planning parallel runs.

For instance, a given search may result in full granules, a series of cross-track scans by the instrument, or a subset of fields-of-regard intersecting a given search perimeter. Different work order handlers may have constraints on whether they can process independent spectra, full scans, or full granules - the “hit-list” provides sufficient information that they can plan effectively, up to and including providing a list of URLs where the raw data can be obtained.

The GeoTemporalSearch interface is summarized in Figure 4.

5.4 *The User Interface*

User interface development was done as a standalone Java-Swing application which could also be built as a McIDAS-V Chooser plug-in. As of this writing, the user interface module supports generation and interaction with Swing forms for job submission and output monitoring and recovery. An additional panel is in the works for executing searches interactively and directing the search results into the inputs of a work order.

Delivery of data to McIDAS-V at the current time is focused on OPeNDAP / NetCDF for data delivery and "ISL" scripts to provide default visualizations. Thus, the processing outputs are best received as NetCDF files, inclusive of unit and provenance metadata, with an accompanying script referencing one or more output sources.

6. APPLICATIONS AND RESULTS

The initial work order module set includes a self-test allowing evaluation of python expressions. Beyond that, we implemented a physical retrieval test pattern involving both MATLAB and FORTRAN modules and extensive ancillary data. Finally, we obtained code for an instrument spectral convolution in MATLAB that allows IASI scans to be convolved to emulate GOES-12 and other satellite sounders’ data.

A major aspect of dealing with prototype algorithms is data format and extraction. Each work order module matched a simple pattern: Cache the software, fetch the data, build a workspace, reformat or extract the input as needed, execute the algorithm, reformat the output to NetCDF, upload results to the cache manager, and finally purge the workspace. As with many parts of this system, making algorithm software packages available as prepackaged images which could be downloaded and cached became the most productive solution, as it lends itself to testing as well as inviting algorithm providers to place explicitly versioned software in accessible download locations. Nonetheless, the integration time for each of these test applications can still require a day of effort at this point in prototyping.

As a secondary result, a toolbox of functions for caching URL content of different schemes (rsync, ftp, http for starters) evolved in support of common functions. We also found it useful to deploy a capable python interpreter environment to include file I/O to and from MATLAB, NetCDF, HDF5, HDF4, and other scientific formats which the algorithm code accepts as inputs and outputs. This core of function is a credit to the scientific Python community and provides a substantial leg up in dealing with the variety of inputs and outputs that are inherent to this domain.

The actual activities of the system for a test case follows the sequence shown in Figure 6.

1. User directs McIDAS-V at the IASI search service and a work order service by their URLs. He/she selects an area and time of interest.
2. A hierarchy of IASI granule subsections is sent to the cache manager and provided to the user for selection as search results. These correspond to cached metadata files with detailed machine-readable information.
3. User directs search results to inputs of a work order - either using full data file URLs, or the URLs of the search results.
4. The Dispatcher invokes the work order's planning function to get list of tasks to execute, and schedules tasks to run on workers - be they local or remote CPUs.
5. Jobs download algorithm software snapshots, ancillary data, and source data to a local workspace and start processing.
6. Jobs complete processing, uploading their output to the cache manager.
7. The user receives notification of work order completion by way of the Dispatcher.
8. The user gets a list of output URLs related to the work order, including data files as well as run reports or visualization scripts bringing outputs together.
9. McIDAS-V loads any visualization script that the work order may have provided, and reads data over OPeNDAP to form a visualization.

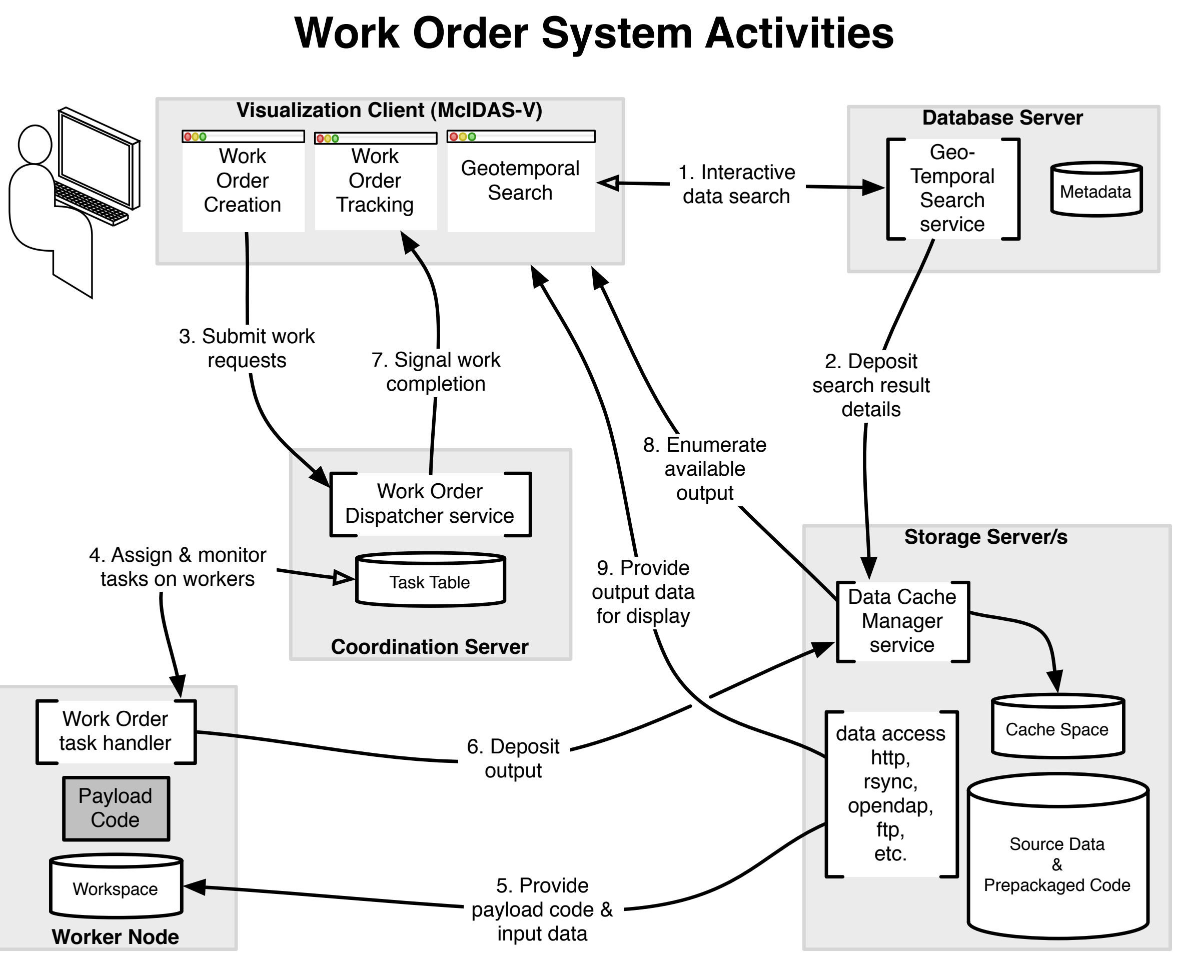
Work Order System Activities
Visualization Client (McIDAS-V)
Work Order Creation
Work Order Tracking
Geotemporal Search
1. Interactive data search
Database Server
Geo-Temporal Search service
Metadata
3. Submit work requests
7. Signal work completion
2. Deposit search result details
8. Enumerate available output
Work Order Dispatcher service
4. Assign & monitor tasks on workers
Task Table
Coordination Server
9. Provide output data for display
Storage Server/s
Data Cache Manager service
Cache Space
Work Order task handler
Payload Code
Workspace
Worker Node
6. Deposit output
data access http, rsync, opendap, ftp, etc.
Source Data & Prepackaged Code
5. Provide payload code & input data

Figure 6. System Activity Diagram.

7. CONCLUSIONS AND FUTURE WORK

As a second prototype, this is a significant boost in functionality and flexibility. It has attained aspects of both a simplified parallel computing framework and an algorithm integration toolkit. However, much work remains to bring the core software to a release state and build dozens of algorithm handlers for end-users. The primary intent of building this system was to improve the scientist's capabilities to analyze algorithms by reducing the time required to locate data, process it and manipulate it.

Other uses are also planned for this service infrastructure, such as a scheduled "nightly build" batching tool for instrument simulations working with (or generating) proxy data.

There is a desire to extend the system to allow products to publish search information and thus permit elaborate loops with search-and-process rule sets to be constructed and run implicitly based on work order profiles.

Further work is needed toward building tools to rapidly deploy database indexing routines for new and experimental datasets. This system only provides a first step in replacing manual data and software gathering. Making it fully useful will doubtless reveal usage patterns which we had not considered.

More extensive support of provenance, including report generation as well as information publication through a web portal or content management system (such as Plone) is of interest as algorithms mature toward production use.

Refining the interface to be intuitive and feature-filled for managing work orders, searches and outputs into a "computational desktop" will be a substantial windfall to usability and acceptance.

Alternate back-end implementations of these services making full use of the substantial efforts of the Grid Computing community would broaden the system capabilities. Likewise, extending the interaction of the Work Order Dispatcher with the client to allow better input validation would make a more modern user interface metaphor than "filling in forms".

Finally, leveraging the literate, unit-sensitive data models inherent in the McIDAS-V and its Java tool-set is likely to bring to light better ways to move data between workers, cache, and client, and combine previously incompatible datasets in support of the advancement of atmospheric science.

8. REFERENCES

Huang, H.L., M. Goldberg. Overview of GOES-R Analysis Facility for Instrument Impacts on Requirements (GRAFIIR) Planned Activities and Recent Progress. Poster, *AMS 88th Annual Meeting, 2008.*

Smuga-Otto, M J, Garcia, R K, Knuteson, R O, Martin, G D, Flynn, B M, Hackel, D (2006), Integrating High-Throughput Parallel Processing Framework and Storage Area Network Concepts Into a Prototype Interactive Scientific Visualization Environment for Hyperspectral Data. *Eos Trans. AGU, 87(52), Fall Meeting Supplementary Abstract A21D-0858, 2007.*

Wallnau, K., S. A. Hissam, R. C. Seacord. Building systems from commercial components. *Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2001.*