LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type	**LIGO-T990097-12 - E**	09/07/2000

The wrapper API's baseline requirements & implementation

Masha Barnes, Kent Blackburn, Albert Lazzarini,
Patrick Brady, Duncan Brown, Jolien Creighton, Alan Wiseman

Distribution of this document:

LIGO
LSC

This is an internal working document
of the LIGO Project.

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (818) 395-2129
Fax (818) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

The LDAS wrapper API's
baseline requirements and implementation

LIGO Laboratory
California Institute of Technology
LIGO Data Analysis System
September 7, 2000

I. Requirements

A. General Requirements:

1. As with all LDAS APIs, the wrapperAPI is being designed and implemented to have a life expectancy equal to that of LIGO (20 years).
2. As with all LDAS APIs, the wrapperAPI will be written to run in the 24 by 7 quasi-realtime environment of the interferometers.
3. The wrapperAPI is responsible for executing the search analysis processes which are based on MPI and executing in the LDAS parallel computing cluster of nodes using an interpreted command language.
4. There will be no limit on the number of searches that can be managed by instances of the wrapperAPI. However, the number of instances expected is of order 6 to 8 search algorithms brought into the wrapper through dynamically loaded shared object libraries (*see the LSC data analysis white paper*) running on the LDAS parallel computing cluster.
   a) The wrapperAPI will support dynamic load balancing. The details of performing load balancing are carried out by the mpiAPI. The wrapperAPI will support the protocol used by the mpiAPI to make decisions and perform load balancing.
   b) The primary function of load balancing is to adjust the nodes associated with the concurrent searches underway in the LDAS Beowulf by reallocating nodes away from searches having the lowest priority level in the system and giving them to searches having higher priority level strictly during times of need in the searches which are allocated to run at the highest priority level.The use of dynamic load balancing will also serve to make minor "steering" adjustments to searches running on the on-site LDAS beowulf systems which will differ ever-so-slightly in areas of clock speed, network interfaces, motherboard design, etc. from the development systems at CIT which are have higher availability for studying performance issues and issues of precise node allocation needs for searches.
5. The wrapperAPI is written entirely in C++. No TCL/TK is used in this LDAS API.

6. The wrapperAPI will be initiated using the mpirun command. This command will be started solely by the mpiAPI (*see LIGO-T990086-E*). The mpiAPI will determine the appropriate values for the mpirun command, as well as the appropriate commands for the wrapperAPI and pass these as command line arguments to mpirun. The wrapperAPI will interpret its own command line arguments which are automatically passed to the wrapperAPI by the mpirun command.
7. The wrapperAPI will process a "segment" of data of finite length. For each new "segment" of data a new wrapperAPI will be started up by the mpiAPI. This "chunk" of data will nominally be provided to the wrapperAPI by the LDAS dataConditionAPI (*see LIGO-T990002-E*).
8. The wrapperAPI is a component of LDAS and therefore must support 24-by-7 operations at the observatories. The wrapperAPI will be restarted with each new segment of data from the interferometers (*or user specified archived data*) being analyzed. This will decrease the risks associated with very small overlooked memory management problems (*memory leaks*) on hardware with finite resources.
9. All messages and data will pass between the search algorithms and LDAS through the wrapperAPI. File I/O will direct communications between the LDAS and search algorithms will not be permitted during normal execution of the wrapperAPI.
10. The scientific results from running the wrapperAPI will be directed to the eventMonitorAPI (*baseline requirements TBD*) using the LDAS ILWD C++ socket communications. Exceptions which involve the use of files for data exchange shall be supported for limited situations such as during early MDC when full LDAS functionality is not available.

B. The TCL/TK Script's Requirements:

1. None applicable to the wrapperAPI since it will not have TCL/TK code. The wrapperAPI is mated toTCL code using the mpiAPI
2. The wrapperAPI deviates somewhat from the normal LDAS API model. It does not have the usual integrated TCL process control layer. Instead this functionality is achieved using through a dedicated socket communication with the mpiAPI. Only ascii based textural messages and commands can be passed through this low bandwidth channel.
3. LDAS parallel job queues and dynamic load balancing will be supervised through the mpiAPI using communications that share the textural message and command interface protocol outlined in the implementation section of this document.

C. The wrapperAPI executable Requirements:

1. The wrapperAPI must be written to LDAS standards.

2. The wrapperAPI must be written in C++ in order to properly use existing LDAS C++ class libraries and data communications standards (ILWDs).
3. The LDAS ILWD data types used for communicating data between the wrapperAPI and the rest of LDAS must be translated into fundamental MPI data types in order to use the MPI send and receive functions by the wrapperAPI. To facilitate this translation, a standard set of C data structures based strictly on the ANSI C standard (*and shared by the C++ ANSI/ISO standard*) will be used internal to the wrapperAPI to facilitate MPI communications.
4. The wrapperAPI will be written to use the MPI 2.0 C++ API, thereby allowing direct interaction with MPI 2.0 exception handling. This API currently support MPICH and LAM. LDAS will only support MPI 1.2 or later from MPICH.
   NOTE: *This MPI 2.0 C++ API is currently under rapid development and provides the first sojourn into the full MPI 2.0 standard. Future versions of the wrapperAPI will migrate towards a full MPI 2.0 design once implementations become available.*
5. The wrapperAPI will be written as an MPI executable, not as an LDAS shared object library. However, the wrapperAPI will make extensive use of other LDAS shared object libraries (*e.g., objectspace, ilwd, general, etc.*).
6. The wrapperAPI will dynamically load a shared object containing the indexed[1] search filter algorithms used for parallel LIGO data analysis. The wrapperAPI will load the dynamically loaded library using the Unix dopen and related Unix functions.
7. The wrapperAPI will be a **parallel** program based on MPICH version 1.2 or later, designed to operate strictly within the LDAS system. Distributed program flow can be supported for the purpose of prototype and when load balancing and progress reporting are not important to LDAS system integrity. (*However, this will result in failure to fully satisfy requirements 6, 9 and 10.*)
8. The wrapperAPI must support dynamic load balancing (***see general requirements above***).
9. Control of the wrapperAPI will be managed through the mpiAPI. This includes load balancing and the possibility of notification to terminate prematurely. In the event of a termination command from the mpiAPI, the wrapperAPI will notify the search algorithms using the standard dynamically loaded shared object function interface (*see applyFilters() in implementation section for details*).

1. The term index here means any mapping between an ordinal series {1,2,3,...} and the metric that is used internally to the search engine to quantify the fraction of the analysis task. In other words index is a metric that indicates "fraction of job completed" when renormalized by its maximum value. Each search has internal freedom in for the definition of this map. The wrapperAPI will progress through the index from 1 to its maximum value.

10. The wrapperAPI must systematically report progress, status, and health of the running wrapperAPI to the LDAS in a timely manner which provides ideally order 10 to 100 communications per execution. This frequency of communications will be tunable from start up parameters. These reports will include progress (*both percent completion and data chunk size in seconds to compute time ratios*), errors & warnings, node usage, and requests for load balancing.
11. The target platform for running the wrapperAPI will be any LDAS Beowulf Cluster. *Current design plans for this cluster are based on Intel Pentium PCs using the Redhat 6.2 or higher operating systems. However, this choice for cluster technology may evolve as the commodity PC market changes. No choices for the wrapperAPI should strictly assume this cluster technology*.
12. The wrapperAPI will divide parallel processing into two general categories:
    a) A single master process that is responsible for communicating commands with mpiAPI and communicating with other LDAS APIs which send and receive data in the form of ILWD objects through LDAS API data sockets. The master process will also act as the central parallel node used by all slave processes in the MPI environment. The master process is also responsible for translating the ILWD data objects into MPI data types necessary for parallel communications.
    NOTE: *All master wrapperAPI processes will run on a single node of the LDAS Beowulf. LDAS will provide a high end SMP master node capable of running all master wrapperAPIs on a common node allowing for better isolation of the Beowulf hardware.*
    b) A collection of slave processes responsible for carrying out the indexed filtering in a parallel manner. Each slave process must dynamically load the indexed analysis algorithm library as part of its initialization. The slaves will communicate analysis results back to the master using MPI data types. The schedule by which results are reported back shall be tunable using mpirun command line options, enabling search codes to better optimize communications cost which local caching of results on the slaves is affordable to the search strategy.
13. The wrapperAPI must provide the following flow control into the dynamically loaded search libraries which:
    a) Initialization ofsearch parameters used in the specific algorithms being carried out.
    b) Generatation of the index map which corresponds to the order in which the wrapperAPI allocates the job across the slaves with the associated portion of the search being carried out on the slaves.
    c) Provision for any pre-processing of data not performed earlier in the data flow but that is to be used in the search algorithms. This includes a mech-

anism for allowing the search algorithm to handle the distribution of input data to the slaves as well as the ability to perform data conditioning on the input data.

d) Looping through the "apply" method of the search over specified indices until the search task is either completed or terminated prematurely.

e) This looping feature also includes a call to memory management tools needed in the dynamically loaded library once per loop.

f) Provision for a single call to clean up after either completionor termination of the search.

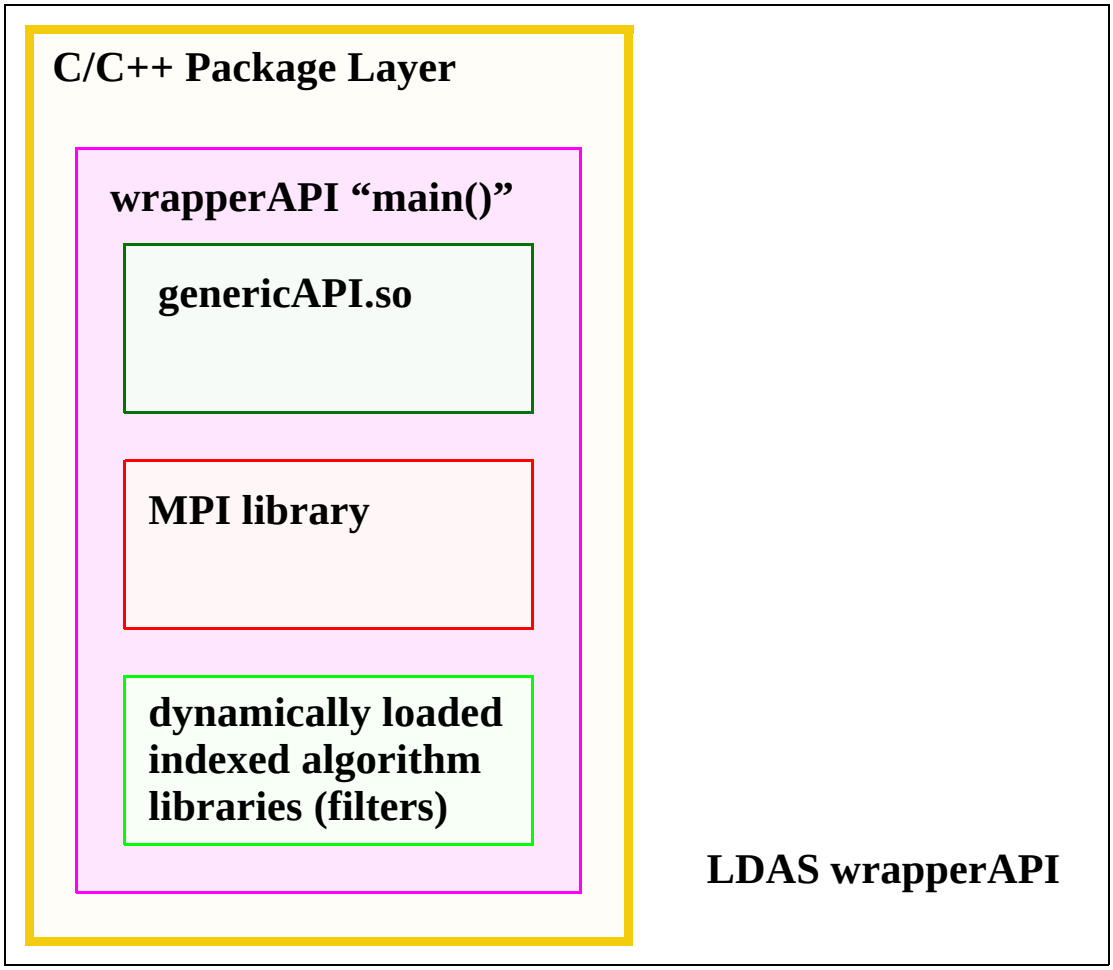
D. WrapperAPI Dynamically Loaded Library Requirements

1. Each type of search will have its own unique dynamically loaded library.
2. Each dynamically loaded library must contain the LDAS specified interface functions that the wrapperAPI resolves through the dynamically loaded library and systematically calls during execution. All calls to interface functions will as specified by the wrapperAPI implementation. These functions will be defined within an **`extern "C" {}`** construct from the wrapperAPI's C++ code (and if necessary by the dynamically loaded library) in order to allow C naming rules to be properly resolved by the C++ compiler name mangling found in the wrapperAPI.
3. Each function exposed to the wrapperAPI by the dynamically loaded library will use an error or warning message string variable. This message string will be nulled out before each call to the any and all interface functions. Any non-null value returned by one of the interface messages will be transmitted by the wrapperAPI using command message communications with the mpiAPI at systematic times (*once per loop iteration*). These messages will be logged into the LDAS logging system by the mpiAPI, posted on the web, and displayed by the controlMonitorAPI (*see LIGO-T000026-E*).
4. The dynamically loaded libraries must support the functional requirements on the wrapperAPI for systematically reporting progress, status, and health of the search to the wrapperAPI in a manner which provides ideally order 10 to 100 communications per execution. This frequency of communications will be tunable from start up parameters.
5. The LDAS system (through the mpiAPI) has the authority to terminate a running wrapperAPI. The wrapperAPI will notify the search when this has occurred allowing the search algorithm to act appropriately. The search algorithm must respond quickly to these notices.
6. Dynamically loaded libraries must support the functional requirement on the wrapperAPI to perform dynamic load balancing.
7. Dynamically loaded libraries must support parallel computation models. Distributed computational models can be supported for prototyping and excep-

tional cases where load balancing and reporting progress are not critical to LDAS system performance.

8. For any dynamically loaded library in which the search algorithm includes MPI communications, these communications must be compatible with MPICH 1.2 or higher. In addition, these libraries must link to the MPI library as a shared object.
9. Each dynamically loaded library will support the LDAS requirements to monitor progress and carry out load balancing of the search algorithm with efficient use of computer performance.
10. Each search algorithm provided by dynamically loaded shared objects for use in the wrapperAPI must be fully validated and verified to the satisfaction of the LIGO Laboratory, the LSC and its software committee. This shall not only include V&V of its scientific and algorithmic implementation but also its capacity to operate within the context of the wrapperAPI and LDAS as a whole.

II. Component Layer Requirements for the wrapperAPI



A. LDAS wrapperAPI:

1. The LDAS wrapperAPI is made up of a single C/C++ layer.
   a) C/C++ Package Layer - this layer is the data engine layer and deals primarily with the binary data and the algorithms and methods needed to manipulate LIGO's data
2. The C/C++ package layer consists of three internal components, developed in C++ and C to take advantage of the higher performance associated with compiled languages which is needed for the types of activities that are being carried out in this layer.
   a) The genericAPI.so - this shared object contains the C++ classes and C interface functions needed to communicate LDAS ilwd data as C++ objects through sockets. It will be linked to the wrapperAPI executable.
   b) The MPI library - this is the Message Passing Interface library used to

communicate MPI based messages and data types between nodes of the parallel process.

c) The dynamically loaded indexed filter algorithm library - this is the library that contains the algorithms and functions necessary to carry out index based parallel filtering (searches) of LIGO's data. It will be loaded as a shared object using the Unix dlopen interface calls.

III. LDAS interface requirements to the wrapperAPI

A. Initialization

1. The LDAS mpiAPI will initiate the wrapperAPI as a stand-alone executable using the mpirun command script. The mpiAPI will be responsible for constructing all command line arguments to the wrapperAPI, this includes options for mpirun as well as options for wrapperAPI.

B. Commands

1. The wrapperAPI will open a Unix socket connection with the mpiAPI's job-state port for the purpose of sending and receiving text commands used to load balance and report status.

C. LDAS Data

1. The wrapperAPI will use the LDAS data sockets for communicating ILWD data with the LDAS system. The functionality to create and manage these data sockets will be derived from the genericAPI's shared object library.

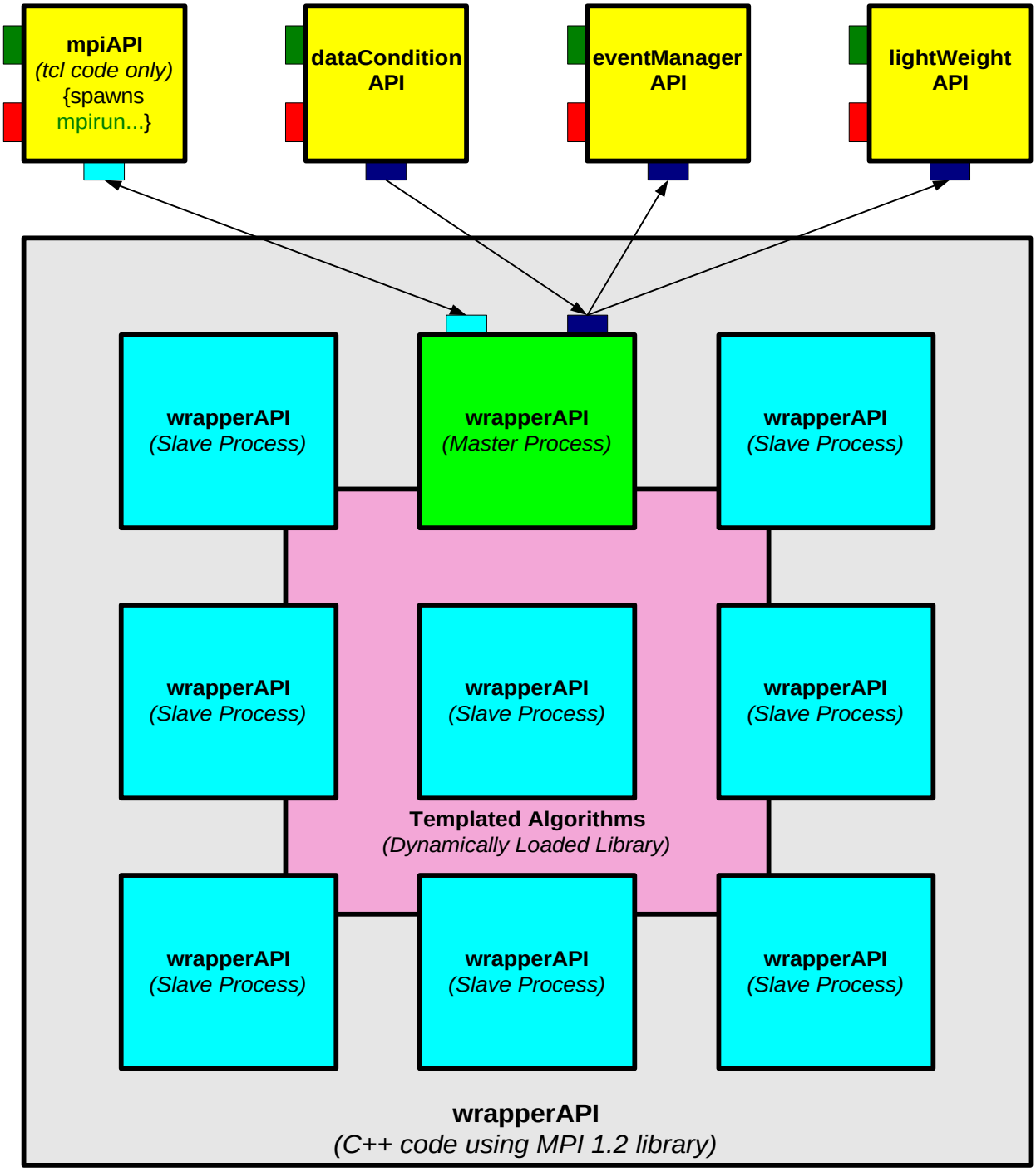
D. MPI

1. The wrapperAPI is a parallel process running on many different nodes of a clustered topology of computers. The software for communicating data and messages between these distinct processes will be the Message Passing Interface (MPI) library. The wrapperAPI will support simplistic parallel processing based on a large set of filter algorithms (indices or templates) being applied in parallel to a single segment of data. The master process will be responsible for sending the data and receiving results of the analysis. The slave processes will carry out the algorithms on the data.

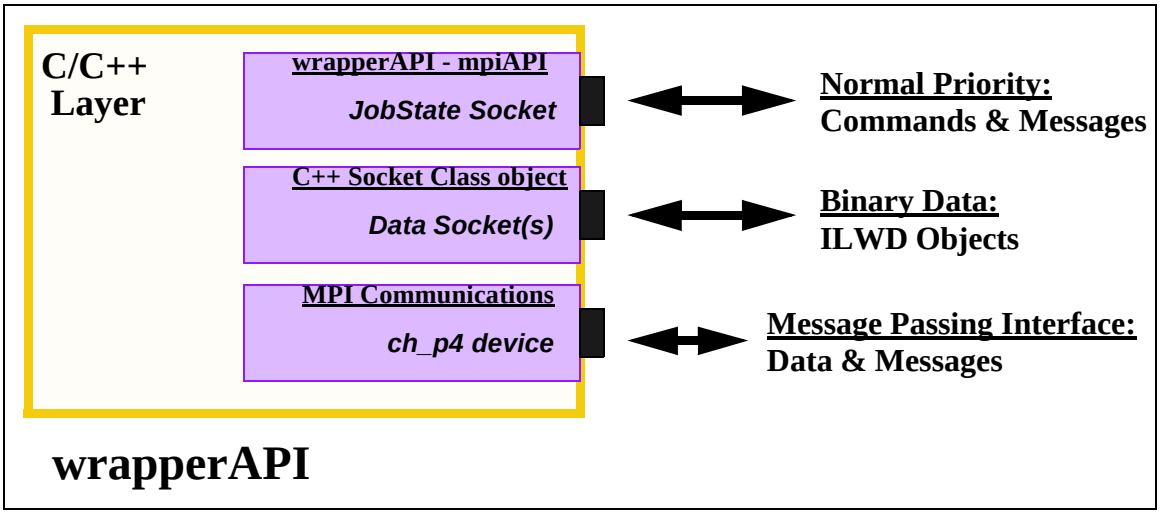
E. Dynamically Loaded Indexed Filter Algorithm DSO Requirements

1. The wrapperAPI will dynamically load all shared object that conforms to the design standard for indexed filter algorithms outlined in this document. The algorithms found in this shared object will be performed within the slave pro-

cess of the parallel process.



IV. Communications Requirements in wrapperAPI



A. Socket Based Communications in wrapperAPI:

1. The wrapperAPI will establish a Unix socket for communicating jobstate commands and messages between the mpAPI and itself. This socket will be managed by the master process of the wrapperAPI.
2. The genericAPI will provide the wrapperAPI with dynamically allocated TCP/IP sockets within the C/C++ layer that is used to communicate LDAS data (*typically binary data*) in the form of streamed binary data or distributed ILWD C++ class objects using the ObjectSpace C++ Component Series Socket Library. This port is commonly referred to as the *Data Socket* to reflect its primary duty in communicating LDAS data sets. Requirements on these sockets are defined by the genericAPI.
3. The MPI library will provide the communications interface (typically the ch_p4 device for clusters) used to share MPI data types and messages between nodes of the parallel process.

V. Required Software Development Tools

A. TCL/TK:

1. TCL is a string based command language. The language has only a few fundamental constructs and relatively little syntax making it easy to learn. TCL is designed to be the glue that assembles software building blocks into applications. It is an interpreted language, but provides run-time tokenization of commands to achieve near to compiled performance in some cases. TK is an TCL integrated (as of release 8.x) tool-kit for building graphical user interfaces. Using the TCL command interface to TK, it is quick and easy to build powerful user interfaces which are portable between Unix, Windows and

Macintosh computers. As of release 8.x of TCL/TK, the language has native support for binary data. The wrapperAPI will not include TCL.

B. C and C++:

1. The C and C++ languages are ANSI standard compiled languages. C has been in use since 1972 and has become one of the most popular and powerful compiled languages in use today. C++ is an object oriented super-set of C which only just became an ANSI/ISO standard in November of 1997. It provided facilities for greater code reuse, software reliability and maintainability than is possible in traditional procedural languages like C and FORTRAN. LDAS software development will be dominated by C++ source code.

C. MPI:

1. The parallel software components of the LDAS will use the public domain version of MPI from MPICH, release 1.2 or greater.
2. The use of MPI code within LDAS will be restricted to the C++ interface bindings and the use of object oriented design technologies whenever possible. The indexed analysis filters and associated functions are not required to be developed using C++ and object oriented design techniques. However, they must support bindings to the core C++ slave processes.

D. SWIG:

1. SWIG is a utility to automate the process of building wrappers to C and C++ declarations found in C and C++ source files or a special *interface file* for API's to such languages as TCL, PERL, PYTHON and GUIDE. LDAS will use the TCL interface wrappers to the TCL extension API's.

E. Make:

1. Make is a standard Unix utility for customizing the build process for executables, objects, shared objects, libraries, etc. in an efficient manor which detects the files that have changed and only rebuilds components that depend on the changed files. The Make facility has been extended using AutoConfig, AutoMake and LibTools, all from the public domain.

F. CVS:

1. CVS is the Concurrent Version System. It is based on the public domain (and is public domain itself) software version management utility RSC. CVS is based on the concept of a software source code repository from which multiple software developers can check in and out components of a software from any point in the development history.

G. Documentation:

1. PERCEPS is a documentation system for C/C++. It generates HTML documents, providing for sophisticated online browsing. The documents are

extracted directly from the source code files. Documents are hierarchical and structured with formatting and references.

2. TclDOC is a documentation system for TCL/TK. It generates structured HTML documents directly from the source code, providing for a similar online browsing system to the LDAS help files. Documents include a hypertext linked table of contents and a hierarchical structured format.

VI. Implementation

A. The mpirun command Implementation:

The basic format of the mpirun command as it will be used by LDAS is the following:
mpirun {*mpirun options***} wrapperAPI {***wrapperAPI options***}**
where mpirun is a command script distributed with MPICH and wrapperAPI is the name of the MPI executable developed by LDAS for parallel computation of index based algorithms.

1. The mpirun options requirements are:
   a) **-np N** which is used to specify the number of processors **N** to used in the parallel computation. The value of **N** is always an integer less than or equal to the total number of processors in the LDAS Beowulf Cluster and is set by mpiAPI and its queue management facilities. If N exceeds the number of processors in the Beowulf Cluster, it will automatically be set to the corresponding value.
   b) **-machinefile /path/file** which is used to identify the list of machine host names used to select our the first N processors required in the previous **-np N** option. The full path and filename for this option must be specified. It is also possible that different instances of the mpirun could use different machinefiles (at the discretion of the mpiAPI). The format of the machinefile is simple:
   ```
   hostname1[:n]
   hostname2[:n]
   ...
   ```
   where the hostname most be of the form return by the unix "hostname" command. This hostname may be followed by an optional ":" and an integer number representing the number of CPUs on that particular host for SMP nodes.
   c) **-nolocal** is an option which specifies to mpirun that the local host is not to be used in the configuration of the parallel processing job. This option may be necessary when the mpiAPI starts a parallel processing job from a host that is not in the core of the Beowulf Cluster (as will be the case in general).
   d) other mpirun options used to test and debug MPI processing will likely

be used during commissioning of the mpiAPI and the wrapperAPI. However, they will not be used in general. Their use must not conflict with the operation of the wrapperAPI and its own set of command line arguments. For more detail on these testing and debugging mpirun command line arguments see the MPICH Users' Guide and Installation Guide.

2. The wrapperAPI options requirements are:

   a) **-nodelist={i-j,k,l,m-n,...}** which is used to specify the subset of nodes to be used by the MPI slave processes in actual calculation of the indexed filters. This list of nodes contains comma delimited node numbers and/or ranges of nodes. All node numbers appearing in this list must be from 0 to N-1, where N is the number of nodes in the commworld specified in the mpirun option **-np** described above. Any integer values in the list greater than N will be ignored.

   b) **-dynlib=/path/libname.a** is used to specify the full (*absolute*) path and file name of the dynamically loaded shared object library containing the indexed filter algorithms. Note: This library must be a shared object library.

   c) **-mpiAPIport={hostname, socketport}** is used to specify the port on the mpiAPI to connect with in order to communicate state information, warnings, errors, job progress, and make requests to balance the load by increasing or decreasing the number of processes associated with the nodelist. The **hostname** parameter specifies the name of the host the mpiAPI is running on and the **socketport** parameter specifies the port the mpiAPI is listening at for the purpose of communications with the wrapperAPI.

   d) **-dataAPI={hostname, socketport}** is used to specify the LDAS API used to provide (serve) data in the ILWD format to the wrapperAPI. Typically this will be the dataConditioningAPI, but others are possible through this argument. Again, the **hostname** specifies the name of the host at with the LDAS API to serve data is running on and the **socketport** parameter specifies the port the data serving LDAS API will be listening at for the purpose of transmitting ILWD formatted data.

   e) **-resultAPI={hostname, socketport}** is used to specify the LDAS API which will receive data products that result from the parallel computation. Again, this data will be shared using the ILWD format. Typically the **resultAPI** will be the eventManagerAPI, however other LDAS APIs may be specified to receive the data products using this argument. The **hostname** parameter specifies the name of the host the receiving API is running on and the **socketport** parameter specifies the port the receiving API is listening at for the purpose of receiving data products from the wrapperAPI.

f) **-dataDistributor={W|WRAPPER || C|CONDITIONDATA}** is used to define the method for distributing input data from the master to the slaves. If the method is W or WRAPPER then all the input data will be sent to all the slaves from the master by the wrapperAPI prior to calling any functions in the dynamically loaded shared object library. If the method is C or CONDITIONDATA then the input data must be distributed to the slaves from the master by the ***conditionData()*** function in the dynamically loaded shared object library. The ***conditionData()*** function on the master will have full control of how the data is distributed, including the option to send unique subsets of data to unique slaves. The pointer to input data returned by ***conditionData()*** on the slaves will be directly passed into the filter algorithm ***applyFilters()*** and must be freed by the call to ***freeFilters()***. NOTE: It is recommended that doLoadBalancing be set to False when the method is C or CONDITIONDATA.

g) **-nodeDutyCycle=N** is the number of indices to be evaluated at each node (in each slave process) per call to the filter algorithm. **N** must be an integer lager than or equal to 1. The wrapperAPI will not allow this number to exceed the total number of indices divided by the number of processors in the comm world. Smaller values of this number allow for more accurate measurements of progress and shorter time intervals for command exchanges between the wrapperAPI and the mpiAPI. Larger values can marginally increase the parallel computation performance by reducing the number of messages passed between master and slave processes.

h) **-slaveReportCycle=N** is the number of calls to the filter algorithm function before sending the results of each index calculated on each slave back to the master process. The default value of N will be 1, meaning that after each call the filter algorithm, the results will immediately be returned to the slave, requiring no local caching of results on the slave. This of course will minimize usage of local memory for storing results but at the same time will maximize the expense of communications overhead. A value of N which is 0 (zero) or larger than the maximum number of indices to be run on the slave will result in all results being cached until completion of the slave's filtering analysis. This of course will provide the most efficient use of communications bandwidth, with a single sending of the results to the master, but will require the most local caching of results data sets. Values of N should be tuned based on the size of result sets and the expense of communications on the provided MPI platform the wrapperAPI is running on.

i) **-communicateOutput={A|ALWAYS || O|ONCE}** is to specify the model of the output structure data object used by the filter algorithm. If the filter algorithm, ***applyFilters()***, can be written such that the content of the output structure returned by the algorithm is unique in content for all calls on all slaves in a particular dynamically loaded shared object, the

the O or ONCE value should be specified, causing the dynamically generated data type used to communicate results data between the slaves and the master to be negotiated only one time in the process. This will tremendously enhance performance and efficiency of communicating results to the master. If the filter algorithm, ***applyFilters()***, can not define a unique results data object between calls, then the A or ALWAYS value should be used. NOTE: In both cases the structure will be analyzed upon return from the filter algorithm call. If O or ONCE is used and the structure has changed, an exception will be generated and the parallel process will terminate to avoid a memory corruption.

j) **-filterparams={a,(b,c,...),d,...}** is used to specify the list of parameters used to control (customize) the parallel filter algorithm. Any parameters within () will be grouped as a single parameter and passed as such. If the designated dynamically loaded library is recognized by the mpiAPI, the values in this list will be validated as being consistent with the expected type, range, and total number for that particular filter library. This will always be the case for LDAS developed dynamically loaded filter libraries. Other libraries which wish to use this mechanism must provide the parameter checks internal to the dynamically loaded library. Numeric parameters **a,b,c,d,...** without decimal places will represent integers. All other numeric parameters will be interpreted as doubles. Everything else will be C strings(char[]).

k) **-realTimeRatio=n.mmmmm** is used to specify the desired ratio of the time required to process the data to the time contained within the data segment. As an example, a value of 0.90 would request that 54 second be used to analyze 60 seconds worth of data.

l) **-doLoadBalance={T|TRUE || F|FALSE}** is to enable or disable load balancing of the parallel process. If the value is T or TRUE then load balancing will be performed as scheduled by the **nodeDutyCycle** command line option. If the value is F or FALSE then no load balancing will be performed. However, the wrapperAPI will still report to the mpiAPI as scheduled by the **nodeDutyCycle** command line option.

B. The wrapperAPI executable implementation:

1. The format of ILWD (Internal Light Weight Data) being received by the wrapperAPI (typically from the dataConditioningAPI) will be of the form of a collection of adc channel data sequences in either the time domain or the frequency domain. A short ASCII ILWD example is given below:

```
<ilwd name='dataConditioningAPI:container' size='2'>
      <ilwd name='XYZ:channel:sequence:primary' size='9'>
          <lstring name='real:domain' dims='4'>TIME</lstring>
          <int_8u name='gps_sec:start_time' units='sec'> 62348734 </int_4u>
          <int_8u name='gps_nan:start_time' units='nanosec'> 0 </int_4u>
```

```
            <int_8u name='gps_sec:stop_time' units='sec'> 62348735 </int_4u>
            <int_8u name='gps_nan:stop_time' units='nanosec'> 0 </int_4u>
            <real_8 name='time:step_size' units='sec'> 0.0625 </real_8>
            <real_4 name='filterX:decimation' dims='1'>1024.000</real_4>
            <real_4 name='methodII:line_removal' dims='3' units='HZ'> 60.0,
            180.0, 360.0 </real_4>
            ...(other filter history)...
            <real_4 name='real:data' dims='16' units='volts'> -0.01, -0.05,
            -0.02, -0.00, 0.01, 0.04, 0.08, 0.11, 0.03, -0.04, -0.07, -0.03,
            0.01, 0.04, 0.06, 0.06 </real_4>
        </ilwd>
        <ilwd name='XYZ:calibration:sequence' size='12'>
            <lstring name='complex:domain' dims='4'>FREQ</lstring>
            <int_8u name='gps_sec:start_time' units='sec'> 62348734 </int_4u>
            <int_8u name='gps_nan:start_time' units='nanosec'> 0 </int_4u>
            <int_8u name='gps_sec:stop_time' units='sec'> 62348738 </int_4u>
            <int_8u name='gps_nan:stop_time' units='nanosec'> 0 </int_4u>
            <real_8 name='start_freq' units='hz'> -2048.0 </real_8>
            <real_8 name='stop_freq' units='hz'> 2048.0 </real_8>
            <real_8 name='freq:step_size' units='hz'> 256.0 </real_8>
            <lstring name='hann:window' size='11'>overlap=15%</lstring>
            ...(other filter history)...
            <real_4 name='real:data' dims='17' units='strain/volt'> -0.31,
            -0.55, -0.12, -0.40, 0.61, 0.24, 0.58, 0.11, 0.13, -0.64, -0.87,
            -0.53, 0.71, 0.84, 0.26, 0.56, 0.91 </real_4>
            <real_4 name='imag:data' dims='17' units='strain/volt'> -0.01,
            -0.05, -0.02, -0.00, 0.01, 0.04, 0.08, 0.11, 0.03, -0.04, -0.07,
            -0.03, 0.01, 0.04, 0.06, 0.06, 0.00 </real_4>
        </ilwd>
        <ilwd name='Wavelet:time-frequency:sequence' size='12'>
            <lstring name='complex:domain' dims='4'>BOTH</lstring>
            <int_8u name='gps_sec:start_time' units='sec'> 62348734 </int_4u>
            <int_8u name='gps_nan:start_time' units='nanosec'> 0 </int_4u>
            <int_8u name='gps_sec:stop_time' units='sec'> 62348735 </int_4u>
            <int_8u name='gps_nan:stop_time' units='nanosec'> 0 </int_4u>
            <real_8 name='time:step_size' units='sec'> 0.0625 </real_8>
            <real_8 name='start_freq' units='hz'> -2048.0 </real_8>
            <real_8 name='stop_freq' units='hz'> 2048.0 </real_8>
            <real_8 name='freq:step_size' units='hz'> 256.0 </real_8>
            <lstring name='uwm-method:wavelet' dims='2' size='14'>
            a=0.30\, b=1.50</lstring>
            ...(other filter history)...
            <real_4 name='real:data' dims='17' units='strain/volt'> -0.31,
            -0.55, -0.12, -0.40, 0.61, 0.24, 0.58, 0.11, 0.13, -0.64, -0.87,
            -0.53, 0.71, 0.84, 0.26, 0.56, 0.91 </real_4>
            <real_4 name='imag:data' dims='17' units='strain/volt'> -0.01,
            -0.05, -0.02, -0.00, 0.01, 0.04, 0.08, 0.11, 0.03, -0.04, -0.07,
            -0.03, 0.01, 0.04, 0.06, 0.06, 0.00 </real_4>
        </ilwd>
        ...(other channels of data)...
</ilwd>
```

In this example two "sequences" of data are sent from the dataConditioningAPI to the wrapperAPI. The first represents channel XYZ in the single precision time domain for a given start time and stop time in GPS seconds and nanoseconds (along with the time interval between time stamps). The channel was decimated by a factor of 1024 using the filter named *filterX*. The next filter applied according to this sequence container is that of line removal

using filter *methodII*. This filter requires an array of real_4 frequencies corresponding to the lines which were removed (*here it is the 60hz, 180hz, and 360hz lines*). Other data conditioning filters may have been applied and they would come here in the container. Next comes the actual sequence of data. It is identified by the name="xxxx:data", where xxxx could be real or imag depending on the domain.

The second sequence container is for a calibration of the adc XYZ. It is data in the frequency domain and is represented by complex data. The frequency range is specified, as well as any filters (*here a hann window function with an overlap of 15%*), followed by the actual data in two distinct ilwd arrays, one for the real component and the other for the imaginary component.

2. It will be the responsibility of the master process to open a ilwd class object socket connection to the LDAS API (*typically the dataConditioningAPI*) server socket specified by the **-dataAPI** command line option and receive the ilwd data object. The master process will interpret (parse) this ilwd data object and reconstruct it as a structured MPI::Datatype for use in MPI communications with the slave processors.
3. It will be the responsibility of the master process to establish ilwd class object socket connections to the LDAS API (typically the eventManagerAPI) server socket specified by the **-resultAPI** command line option and send ilwd table data objects (*along with any optional ilwd sequence data in containers of the type outlined in item **4** above*) when data is ready to be transferred. The master will construct these ilwd objects out of the results received from the slaves as MPI data types.
4. The master process will also be responsible for communicating all state information, warnings, errors, job progress, and make requests to balance the load by increasing or decreasing the number of processes associated with the nodelist. This informations will be communicated using simple text strings sent to the mpiAPI's listening socket designated by the **-mpiAPIport** command line option using just a simple unix socket connection. In general, multiple commands may be sent at a time in a set to the mpiAPI, each separated by a newline **'\n'** character and each set using the same request ID #. Supported command syntax which the wrapperAPI sends to the mpiAPI is as follows:
   a) **"#:request add N"** where **#** is the request ID (*an incremental counter starting at 1*) and **N** is the number of nodes the wrapperAPI would like to add to the process space associated with the current comm world. The mpiAPI will respond to this request with one of the following four forms of syntax (*NOTE - a request to add may be answered with an order to subtract nodes or even to kill the parallel job*):
      (1) **"#:add N {i-j,k,l,m-n,...}"** where **#** is the original request ID and **N** may or may not agree with the requested number of nodes and is

zero or larger, but can not exceed the comm world. The list in square brackets consists of the actual **N** nodes involved in the **add**.

(2) **"#:sub N {i-j,k,l,m-n,...}"** where # is the original request ID and **N** may or may not agree with the requested number of nodes and is zero or larger, but can not exceed the comm world. The list in square brackets consists of the actual **N** nodes involved in the **sub**.

(3) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

(4) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

b) **"#:request sub N"** where # is the request ID (*an incremental counter starting at 1*) and **N** is the number of nodes the wrapperAPI would like to subtract from the process space associated with the current comm world. The mpiAPI will respond to this request with one of the following four forms of syntax (*NOTE - a request to subtract may be answered with an order to add nodes or even to kill the parallel job*):

(1) **"#:sub N {i-j,k,l,m-n,...}"** where # is the original request ID and **N** may or may not agree with the requested number of nodes and is zero or larger, but can not exceed the comm world. The list in square brackets consists of the actual **N** nodes involved in the **sub**.

(2) **"#:add N {i-j,k,l,m-n,...}"** where # is the original request ID and **N** may or may not agree with the requested number of nodes and is zero or larger, but can not exceed the comm world. The list in square brackets consists of the actual **N** nodes involved in the **add**.

(3) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

(4) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

c) **"#:warning {list of warning messages}"** where # is the request ID (*an incremental counter starting at 1*) and **warning** reports that a warning level exception has occurred at some level of the wrapperAPI which is described by the messages contained in the **list**. Typically warnings will be used to indicate that a non-fatal condition exists in the wrapperAPI's execution. The mpiAPI log this warning message using the standard LDAS logs file system and then will respond to this request with one of the following forms of syntax:

(1) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

(2) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

d) **"#:error {list of error messages}"** where # is the request ID (*an incremental counter starting at 1*) and **error** reports that a error level exception has occurred at some level of the wrapperAPI which is described by the messages contained in the **list**. Typically error will be used to indicate that a fatal condition exists in the wrapperAPI's execution. The mpiAPI logs this error message using the standard LDAS log file system and then will respond to this request with one of the following forms of syntax:

   (1) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

   (2) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

e) **"#:progress nnn.mm%"** where # is the request ID (*an incremental counter starting at 1*) and **nnn.mm%** is the percent complete for the wrapperAPI's parallel process job. The mpiAPI logs this error message using the standard LDAS log file system and then will respond to this request with one of the following forms of syntax:

   (1) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

   (2) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

f) **"#:using N {i-j,k,l,m-n,...} nodes out of the M available in comm world"** is the default "nominal" command where # is the request ID (*an incremental counter starting at 1*) and N is the number of nodes being actively used (more specifically the **N** found in the list **[i-j,k,l,m-n,...]**) from the **M** available in the comm world. The mpiAPI logs this warning message using the standard LDAS log file system and then will respond to this request with one of the following forms of syntax:

   (1) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

   (2) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

g) **"#:projected ratio n.mmmmm"** where # is the request ID (*an incremental counter starting at 1*) and **n.mmmmm** is the ratio of the projected time to completion to the amount of data being analyzed. The mpiAPI logs this error message using the standard LDAS log file system and then will respond to this request with one of the following forms of syntax:

   (1) **"#:cont"** where # is the original request ID and **cont** instructs the wrapperAPI to continue processing without any changes.

   (2) **"#:kill"** where # is the original request ID and **kill** instructs the wrapperAPI to cleanly shutdown all mpi parallel code and exit.

The wrapperAPI will typically send a subset of these commands to the mpi-API upon completion of each cycle of data through the slave processes.

h) The set of commands will consist of either of the following sets:

(1) During regular processing:

one command from **a)** or **b)** or **f)**

plus

command **c)** if warnings occurred

plus

command **d)** if errors occurred

plus

command **e)** and command **g)**.

(2) At the completion all analysis:

command **e)** with progress at 100.00%

plus

command **c)** if warnings occurred.

5. The wrapperAPI will provide a method to estimate the number of nodes needed to run the parallel process in real time and calculate the load balancing request as integer nodes, such that the projected ratio is less than or equal to **realTimeRatio**, while remaining as close to **realTimeRatio** as possible. This in itself requires that the wrapperAPI be able to extract the length of the data sequence in terms of collection time, while also measuring progress on analyzing the data in wall clock time.

C. WrapperAPI Dynamically Loaded Library Interface

1. The wrapperAPI will load all indexed analysis algorithms from dynamically loaded libraries from the local Beowulf file space. Each type of search will have its own dynamically loaded library. This will allow each mpirun command to be associated with a particular type of search by the particular dynamically loaded library assigned by the command line arguments (see **-dynlib** command line option above). The wrapperAPI will load the dynamically loaded library using the Unix dopen and related functions:

```
SYNOPSYS
   #include <dlfcn.h>

   void *dlopen (const char *filename, int flag);
   const char *dlerror( void );
   void *dlsym( void *handle, char *symbol );
   int dclose (void *handle);

   Special symbols _init, _fini.
```

The flag variable used in the dlopen call must not include the

RTLD_GLOBAL flag value. See the Unix (Linux) manpages for more details on the use of dlopen.

2. Each slave process will dlopen the dynamically loaded library specified by the command line option. Each dynamically loaded library must contain six C functions used to interface the wrapperAPI with the dynamically loaded library. All six of these required functions will be called in order by each slave. These functions will be defined within an **`extern "C" {}`** from the wrapperAPI's C++ code. Each function will use an error or warning message string variable. This message string will be nulled out before each call to one of the five interface functions. Any non-null value returned by one of the interface messages will be cached by the wrapperAPI for command message communications with the mpiAPI at a convenient time. These messages will be logged into the LDAS logging system by the mpiAPI:

   a) The first required C function is used to initialize the dynamically loaded library and to configure the parameterization as defined by the command line option **-filterparams**. It is called once by the master and each slave in the MPI_COMM_WORLD:

```
SYNOPSIS

extern "C" {

#include "wrapperInterface.h"

CHAR *initMessage = 0;
free(initMessage);
INT4 initFilters( INT4 argc, CHAR* argv[], CHAR** initMessage );
}
```

   where **`argc`** is a count of the number of parameter arguments being passed into the dynamically loaded library from the -filterparams command line option and **`argv`** is an array of pointers to the -filterparams command line options. Note that the argv[0] pointer will store the string "-filterparams". This will allow initFilters() to verify that the options are associated with the appropriate command line option. As an example, if the command line contained -filterparams=[1.0, 3.0, 10.0], the value of argc would be 4 and argv[0] would point to "-filterparams", argv[1] would point to "1.0", argv[2] would point to "3.0", and argv[3] would point to "10.0". The initFilter() function will be responsible for parsing the character values pointed to by argv into numerical values. As noted earlier all floating point values MUST have a decimal point. Non numerical values are allowed by this mechanism, however they should be discouraged.

   The **initFilters()** function returns a 0 (zero) value if it is successful in parsing and interpreting the values from argv (including any and all range checking on these values). In the event of an error, the initFilter() func-

tion must return a 1 (one) and assign a unique error message to the **initMessage** variable. In the event that a warning condition is established, the value returned by initFilter shall be -1 and the corresponding warning message will be stored in the **initMessage** variable. Any warning or error level message stored in **initMessage** will be logged under in the LDAS wrapperAPI log file, along with the GPS time, jobID, hostname, node number and the initFilter() function name. Any memory associated with **initMessage** will be freed before the call to initFilter().

The values parsed from the **argv** pointers must be stored internally in the global variable space of the dynamically loaded library if they are to be used by other functions within the dynamically loaded library.

b) The second required function is used to calculate the total number of indexed filters that will be used in the search algorithms contained within the dynamically loaded library. It is called once by the master and each slave in the MPI_COMM_WORLD. Filters will be called using an index counting scheme that ranges from 1 to the maximum specified by the value returned within **numberIndices**. It may also be used to internally construct a one-to-one identification map between the sequential index values and a specific set of parameter values used by the indexed filter algorithms:

```
SYNOPSIS

extern "C" {

#include "wrapperInterface.h"

UINT4 numberIndices = 0;
CHAR* indexMessage = 0;

free(indexMessage);
INT4 indexFilters( UINT4 *numberIndices, CHAR** indexMessage );
}
```

The nominal return value for indexFtilters() will be 0 (zero). If the indexFilters() function has an error then it must return with a value of 1 (one) and assign a unique error message to **indexMessage**. A warning may be reported by a return value of -1 from indexFilters() and providing a warning message within **indexMessage**. This function will be guaranteed to be called by each node or processes of the wrapperAPI before the first indexed search filter is called. Any warning or error level message stored in **indexMessage** will be logged under in the LDAS wrapperAPI log file, along with the GPS time, jobID, hostname, node number and the indexFilters() function name. Any memory associated with **indexMessage** will be freed before the call to indexFilters().

c) The third required function is used to carry out any further pre-condition-

ing of data which may not be possible for the LDAS dataConditionAPI. It is called by each slave process. It will have write privileges on the input data structure:

```
SYNOPSIS

extern "C" {

#include "wrapperInterface.h"

CHAR* conditionMessage = 0;
inPut data[];
MPI_Comm* comm;

free(conditionMessage);
INT4 conditionData( inPut* data, CHAR** conditionMessage,
MPI_Comm* comm );
}
```

The conditionData() function is called with the **`inPut`** data structure which contains the data to be further conditioned. **NOTE:** It is highly recommended that use of this function be held to a minimum. Ideally any functionality present in this function should be migrated upstream into the dataConditionAPI. Hence, under most circumstances it simply returns successfully as described below without modifying the **`inPut`** data structure. The justification for this being that it is less expensive to condition the data upstream once than to condition it a total of N times on each of the slave processes. The **conditionData()** function should nominally never overwrite data found in the **inPut** data structure, only extend it with new data local to the node it occupies. It may be that certain types of analysis require that the data be customized local to each node. Under these circumstances the **conditionData()** function may alter the inPut data structure so long as the analysis performed in **applyFilters()** on each node can support this customizing for the duration of the wrapperAPI's execution.

The **conditionData()** function will return an integer value of 0 if successful and an integer value of 1 along with a unique error message in the **`conditionMessage`** variable. If a warning occurs the **conditionData()** function will return (***with results***) a value -1 along with the warning messages stored in the **`conditionMessage`** variable. Any warning or error level message stored in **`conditionMessage`** will be logged under in the LDAS wrapperAPI log file, along with the GPS time, jobID, hostname, node number and the conditionData() function name. Any memory associated with **`conditionMessage`** will be freed before each call to conditionData().

The local and global node information are also passed into the **conditionData()** function using the MPI communicator structure **comm**. Using this

structure it is possible for **conditionData()** to identify the local node, the total nodes in the communicator and to make node-to-node communications if this is the desired way to split up input data. NOTE: It is important to consider in the design of **conditionData()** that it will be called multiple times as part of a loop within the wrapperAPI. Also, each time that it is called the MPI communicator may have been modified to include a different set of nodes. Thus, the **conditionData()** may require static internal data to recognize when to repeat steps or when to redo steps associated with conditioning data, including node-to-node communications which may be used to pass data to **indexFilters()** instead of using the conventional path through the **inPut** data structure passed by the call to **indexFilters()**. This may be useful, for example, to distribute local data in a distributed FFT algorithm.

d) The fourth required function is the parallel search engine. It is used to apply a particular set of index filter parameters to the data and to return the results of the filtration algorithms. It will typically be called repeatedly in a loop on each slave process until all indexed filters have been analyzed. As such it has a highly specialized set of input and output structures which must be general enough to carry out index filter gravitational wave searches from interferometer data:

```
SYNOPSIS

extern "C" {

#include "wrapperInterface.h"

INT4 beginIndex;
INT4 endIndex;
CHAR* filterMessage = 0;
inPut data[];
outPut* result[1+endIndex-beginIndex];
BOOLEAN finalCall;
MPI_Comm* comm;

free(filterMessage);
INT4 applyFilters( INT4 beginIndex, INT4 endIndex,
const inPut* data, outPut** result, CHAR** filterMessage,
BOOLEAN finalCall, MPI_Comm* comm );
}
```

The **applyFilters()** function is called only by the active slaves over and over again in a loop with two integers **`beginIndex`** and ending at **`endIndex`** until all indices have been analyzed. These two variable can be identical, signifying that only one index will be applied to the data found in the inPut data structure prior to returning the results, otherwise each index will be analyzed using the data in **`inPut`** before returning. The **applyFilters()** function must, internal to the dynamically loaded library,

associate this index to a set of physical parameters associated with the filter model. The **applyFilters()** function must cast the **`inPut`** into LAL standard data structures appropriate for the internal LAL algorithms being used. The **applyFilters()** function must cast results into the **`outPut`** structure specified above.

The typedef **`inPut`** structure will be an array of data sequences associated with pre-conditioned interferometer channel date. In most cases only one sequence containing the pre-conditioned strain signal from the interferometer will be contained in this structure. However, using the array construct, more complex filters using multi-channel data sequences are supported. **NOTE:** The **applyFilters()** function must not modify the contents of this typedef **`inPut`** structure as it may be needed for further analysis by the slave process using a different set of template indices (unless subsequent calls to **applyFilters()** expect the input data to be modified).

The typedef **`outPut`** structure will be an array of pointers large enough to hold all results from all indices (1+**endIndex**-**beginIndex**) analyzed during the call to the **applyFilters()** function. The **applyFilters()** function will be responsible of allocating needed memory for the **outPut**.

The results of the **applyFilters()** function are stored in the **`results`** array of **`outPut`** typedef structures. The wrapperAPI is responsible for evaluating the **`significant`** attribute of each element of the **`results`** array and in the event it is **`TRUE`** making a deep copy (on the wrapperAPI side) of the contents of the **`dataBase`** doubly linked list typedef structure in the memory space of the wrapperAPI, allowing the **applyFilters()** function to reuse this variable space. NOTE: This includes making a copy of the **`optional`** array of sequence typedef structure attributes if they are not NULL.

The **applyFilters()** function will return an integer value of 0 if successful and an integer value of 1 along with a unique error message in the **`filterMessage`** variable. If a warning occurs the **applyFilters()** function will return (***with results***) a value -1 along with the warning messages stored in the **`filterMessage`** variable. Any warning or error level message stored in **`filterMessage`** will be logged under in the LDAS wrapperAPI log file, along with the GPS time, jobID, hostname, node number and the **applyFilters()** function name. Any memory associated with **`filterMessage`** will be freed before each call to **applyFilters()**.
The LDAS system has the authority to end a wrapperAPI job in the midst of execution. To allow the underlying search algorithm to gracefully recongnize this impending termination, the **applyFilters()** will always be called on its last pass with the boolean flag **finalCall** set to TRUE.

The local and global node information are also passed into the **applyFilters()** function using the MPI communicator structure **comm**. Using this

structure it is possible for **applyFilters()** to identify the local node, the total nodes in the communicator and to make node-to-node communications if this is the desired way to split up the filter analysis. NOTE: It is important to consider in the design of **applyFilters()** that it will be called multiple times as part of a loop within the wrapperAPI. Also, each time that it is called the MPI communicator may have been modified to include a different set of nodes. Thus, the **applyFilters()** may require static internal data to recognize when to repeat steps or when to redo steps associated with analyzing data, including node-to-node communications. This may be useful, for example, to distribute local data in a distributed FFT algorithm.

e) The fifth required function is used to free the memory associated with the array of output structures returned by the **applyFilters()** function above. It must be called to guarantee that the associated LAL memory deallocation routines are used to free memory previously allocated with LAL memory allocation routines.

```
SYNOPSIS

extern "C" {

#include "wrapperInterface.h"

outPut* result[1+endIndex-beginIndex];
CHAR* freeMessage = 0;

free(freeMessage);
INT4 freeOutput( outPut** result, CHAR** freeMessage );
}
```

The **freeOutput()** function will go through and free unneeded memory from each non-NULL array element in the array of **outPut** structures. NOTE: if state information was stored in the **outPut** for subsequent calls to the **applyFilters()** function, it should not be freed. This function will return an integer value of 0 (zero) upon successfully deallocating memory and a 1 if the memory deallocation fails along with an error message in the **`freeMessage`** variable. The value of -1 will be returned if a warning occurs along with a message in **`freeMessage`**. Any warning or error level message stored in **`freeMessage`** will be logged under in the LDAS wrapperAPI log file, along with the GPS time, jobID, hostname, node number and the **freeOutput()** function name.

f) The last required function is used to free up any dynamically allocated memory that may be associated with the indexed filter search algorithms found in the dynamically loaded library. It is called once at the completion of all calls to **applyFilters()** by the master and all slaves of the MPI_COMM_WORLD.

```
SYNOPSIS
```

```
extern "C" {

#include "wrapperInterface.h"

CHAR* freeMessage = 0;

free(freeMessage);
INT4 freeFilters( CHAR** freeMessage );
}
```

The specifics of this function will depend on the internal workings of the algorithms found in each particular dynamically loaded search library. For example, if a one-to-one identification map was dynamically constructed as an index, then it must be freed upon completion of the search. This function will return an integer value of 0 (zero) upon successfully deallocating memory and a 1 if the memory deallocation fails along with an error message in the **`freeMessage`** variable. The value of -1 will be returned if a warning occurs along with a message in **`freeMessage`**. Any warning or error level message stored in **`freeMessage`** will be logged under in the LDAS wrapperAPI log file, along with the GPS time, jobID, hostname, node number and the freeFilters() function name.

g) The definitions for these interface functions and data types outlined above will be included into code via the **`wrapperInterface.h`** file. This header file will contain valid ANSI C syntax. Also, this header file must be guarded and must be included within an **`extern "C" {}`** block for C++ code as previously illustrated. The structures defined within **`wrapperInterface.h`** are given below:

```
/* prevent multiple inclusions of header file */

#ifndef WRAPPER_INTERFACE_H
#define WRAPPER_INTERFACE_H

/* include this file to get interface datatypes */
#include "wrapperInterfaceDatatypes.h" /* datatype header file */

/* ANSI C prototypes for four interfacing functions */

#ifdef __cplusplus
extern "C" {
#endif

/* LDAS_BUILD must define these as external resolved functions */
#ifdef LDAS_BUILD
#define LDAS_EXTERN extern
#else
#define LDAS_EXTERN
#endif
```

```
LDAS_EXTERN INT4 initFilters( INT4 argc, CHAR* argv[],
CHAR** initMessage );

LDAS_EXTERN INT4 indexFilters( UINT4* numberIndices,
CHAR** indexMessage );

LDAS_BUILD INT4 conditionData( inPut* data,
CHAR** conditionMessage, MPI_Comm* comm );

LDAS_BUILD INT4 applyFilters( INT4 beginIndex,
INT4 endIndex, const inPut* data, outPut* result,
CHAR** filterMessage, BOOLEAN finalCall, MPI_Comm* comm );

LDAS_BUILD INT4 freeFilters( CHAR** freeMessage );

#undefine LDAS_BUILD

#ifdef __cplusplus
}
#endif

#endif
```

This file includes the header file **wrapperInterfaceDatatypes.h** which contains all the data types used by the interface functions. It contents is given below:

```
/* prevent multiple inclusions of header file */

#ifndef WRAPPER_INTERFACE_DATATYPES_H
#define WRAPPER_INTERFACE_DATATYPES_H

#ifdef __cplusplus
extern "C" {
#endif

typedef enum {timeD, freqD, bothD} domain;

typedef enum { boolean_1u, char_s, char_u,
int_2s, int_2u, int_4s, int_4u, int_8s, int_8u,
real_4, real_8, complex_8, complex_16
} datatype;

/* include this file to get LAL datatypes */
#include <mpi.h>
#include "LALAtomicDatatypes.h" /* LAL header file */

typedef union { /* these pointer types MUST exist in LAL! */
BOOLEAN *boolean; /* pointer to BOOLEAN type */
```

```
CHAR *chars; /* pointer to CHAR */
UCHAR *charu; /* pointer to UCHAR */
INT2 *int2s; /* pointer to INT2 */
UINT2 *int2u; /* pointer to UINT2 */
INT4 *int4s; /* pointer to INT4 */
UINT4 *uint4u; /* pointer to UINT4 */
INT8 *int8s; /* pointer to INT8 */
UINT8 *int8u; /* pointer to UINT8 */
REAL4 *real4; /* pointer to REAL4 */
REAL8 *real8; /* pointer to REAL8 */
COMPLEX8 *complex8; /* pointer to COMPLEX8 */
COMPLEX16 *complex16; /* pointer to COMPLEX16 */
} dataPointer; /* union supporting pointer type checking */

typedef struct {
UINT8 numberSamples; /* no. of data samples in interval */
UINT8 startSec; /* GPS start time in seconds */
UINT8 startNan; /* GPS start time in nanoseconds */
UINT8 stopSec; /* GPS stop time in seconds */
UINT8 stopNan; /* GPS stop time in nanoseconds */
REAL8 timeStepSize; /* uniform step size in seconds */
} gpsTimeInterval; /* time domain interval */

typedef struct {
UINT8 numberSamples; /* no. of data samples in interval */
UINT8 gpsStartTimeSec; /* GPS start time in seconds */
UINT8 gpsStartTimeNan; /* GPS start time in nanoseconds */
UINT8 gpsStopTimeSec; /* GPS start time in seconds */
UINT8 gpsStopTimeNan; /* GPS start time in nanoseconds */
REAL8 startFreq; /* starting frequency in hertz */
REAL8 stopFreq; /* ending frequency in hertz */
REAL8 freqStepSize; /* uniform step size in hertz */
} frequencyInterval; /* frequency domain interval */

typedef struct {
UINT8 numberSamples; /* no. of data samples in interval */
UINT8 gpsStartTimeSec; /* GPS start time in seconds */
UINT8 gpsStartTimeNan; /* GPS start time in nanoseconds */
UINT8 gpsStopTimeSec; /* GPS stop time in seconds */
UINT8 gpsStopTimeNan; /* GPS stop time in nanoseconds */
REAL8 startFreq; /* starting frequency in hertz */
REAL8 stopFreq; /* ending frequency in hertz */
REAL8 timeStepSize; /* uniform step size in seconds */
REAL8 freqStepSize; /* uniform step size in hertz */
} timeFreqInterval; /* frequency domain interval */

typedef union {
gpsTimeInterval dTime; /* time domain interval info */
frequencyInterval dFreq; /* frequency domain interval info */
timeFreqInterval dBoth; /* time+frequency domain interval info */
} interval;
```

```
#define maxHistoryName 64
#define maxHistoryUnits 64

typedef struct dcHistoryTag {
struct dcHistoryTag *previous; /* previous data cond. filter */
CHAR name[maxHistoryName]; /* data conditioning filter name */
CHAR units[maxHistoryUnits]; /* data conditioning filter units */
datatype type; /* data type for column */
UINT4 numberValues; /* no. rows to add to column */
dataPointer value; /* pointer to table's column data */
struct dcHistoryTag *next; /* next data cond. filter */
} dcHistory; /* this is a bi-directional linked list */

#define maxStateName 64

typedef struct stateVectorTag {
struct stateVectorTag *previous; /* previous state vector */
CHAR stateName[maxStateName]; /* name of state*/
multiDimData *store; /* reuseable state vector data store */
struct stateVectorTag *next; /* next state vector */
} stateVector;

typedef struct {
CHAR name[256] /* name of the data in dataPointer */
CHAR units[256] /* comma separated units of the data */
domain space; /* either time, frequency or both domain */
datatype type; /* type of data in pointer */
interval range; /* epoch of time/frequency for data */
UINT4 numberDimensions; /* no. of dimensions in data */
UINT4 dimensions[]; /* no. of elements along each dimension */
dcHistory *history; /* data conditioning history */
dataPointer data; /* pointer to multi-dimensional data */
} multiDimData;

typedef struct {
UINT4 numberSequences; /* number of data channels in inPut */
stateVector *states; /* input state vector information */
multiDimData sequences[]; /* array of conditioned data */
} inPut;

typedef enum { binaryInspiral, ringDown, periodic, burst,
stocastic, timeFreq, instrumental, protoType, experimental
} catagory; /* astrophysical/instrumental search catagories */

#define dbNameLimit 19 /* Note DB2 limits names to 18 letters */

typedef struct dataBaseTag {
struct dataBaseTag *previous; /* previous table data set */
CHAR tableName[dbNameLimit]; /* name of LDAS table */
CHAR columnName[dbNameLimit]; /* column name in LDAS table */
```

```
datatype type; /* data type for column */
UINT4 numberRows; /* no. rows to add to column */
dataPointer rows; /* pointer to table's column data */
struct dataBaseTag *next; /* next table data set */
} dataBase; /* this is a bi-directional linked list */

typedef struct {
INT8 indexNumber; /* number of index results in outPut */
catagory search; /* type of astrophysical/instrumental search */
BOOLEAN significant; /* signals that require post-processing */
stateVector *states; /* output state vector information */
dataBase *results; /* filter results to be ingested into DB */
multiDimData *optional; /* optional sequences (1 per index) */
} outPut;

#ifdef __cplusplus
}
#endif

#endif
```

This **wrapperInterface.h** header file learns about the LAL standard datatypes by including the **LALAtomicDatatypes.h** header file. This header file must at a minimum contain the following definitions:

```
#ifndef _LALATOMICDATATYPES_H
#define _LALATOMICDATATYPES_H

#ifdef LDAS_BUILD
#include "LDASConfig.h"
#else
#include "LALConfig.h"
#include "LALRCSID.h"
RRCSID( LALATOMICDATATYPESH, "$Id: LALAtomicDataTypes.h,
v 1.1 2000/04/20 20:02:33 jolien Exp $")
#endif

typedef char CHAR;

typedef unsigned char UCHAR;

typedef unsigned char BOOLEAN;

#if SIZEOF_SHORT == 2
typedef short INT2;
typedef unsigned short UINT2;
#elif SIZEOF_INT == 2
typedef int INT2;
typedef unsigned int UINT2;
#else
```

```
#error "ERROR: NO 2 BYTE INTEGER FOUND"
#endif

#if SIZEOF_INT == 4
typedef int INT4;
typedef unsigned int UINT4;
#elif SIZEOF_LONG == 4
typedef long INT4;
typedef unsigned long UINT4;
#else
#error "ERROR: NO 4 BYTE INTEGER FOUND"
#endif

#if SIZEOF_LONG == 8
typedef long INT8;
typedef unsigned long UINT8;
#elif SIZEOF_LONG_LONG == 8
typedef long long INT8;
typedef unsigned long long UINT8;
#else
#error "ERROR: NO 8 BYTE INTEGER FOUND"
#endif

#if SIZEOF_FLOAT == 4
typedef float REAL4;
#else
#error "ERROR: NO 4 BYTE REAL FOUND"
#endif

#if SIZEOF_DOUBLE == 8
typedef float REAL8;
#else
#error "ERROR: NO 8 BYTE REAL FOUND"
#endif

typedef struct {
  REAL4 re;
  REAL4 im;
} COMPLEX8;

typedef struct {
  REAL8 re;
  REAL8 im;
} COMPLEX16;

#endif
```

Notice that this **LALAtomicDatatypes.h** header file will include the LDAS **LDASConfig.h** header file which is generated by the LDAS autoconfig scripts at configuration time when **LDAS_BUILD** is defined

(by LDAS) and will include the **LALConfig.h** header file which is generated by the LAL autoconfig scripts at configuration time otherwise. In the event that this file is not available, the necessary definitions found in this file for both SPARC Solaris and Intel Pentium Linux computers using the GCC 2.95.2 compiler can be placed in a mock-up ***config.h*** file which looks like the following:

```
/* for Intel Pentium Linux and SPARC Solaris using */
/* GCC version 2.95.2 */

#ifndef TypeConfigH
#define TypeConfigH

/* The number of bytes in a double.  */
#define SIZEOF_DOUBLE 8

/* The number of bytes in a float.  */
#define SIZEOF_FLOAT 4

/* The number of bytes in a int.  */
#define SIZEOF_INT 4

/* The number of bytes in a long.  */
#define SIZEOF_LONG 4

/* The number of bytes in a long long.  */
#define SIZEOF_LONG_LONG 8

/* The number of bytes in a short.  */
#define SIZEOF_SHORT 2

#endif
```

This mock-up file may not contain any other definitions that conflict with LDAS. In the integrated build of the wrapperAPI, the config.h file generated by the LDAS autoconfig scripts shall be used to define these sizes. NOTE: The SIZEOF_UNSIGNED_* are not needed as they are guaranteed to be consistent with signed sizes on the LDAS target platforms at this time (early 2000).

h) The dynamically loaded library (dll) shared object file will define the interface functions. The top level C source file **LALWrapperInterface.c** for any dll shared object which provides the interface definition will contain the following:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <setjmp.h>
#include "wrapperInterface.h"
```

```
#include "LALWrapperInterface.h"
#include "LALmalloc.h"

NRCSID( LALWRAPPERINTERFACEC,"$Id: LALWrapperInterface.c$ );

#define STRINGIFY_HELPER( a ) #a
#defind STRINGIFY( a ) STRINGIFY_HELPER( a )
#define FILELINEID ", file: " __FILE__ \
", line: "stringify( __LINE__ ) \
", $Id: LALWrapperInterface.c$"

int debuglevel = 0;
enum {warning = -1, Nominal = 0, Error = 1 };

static INT4 stringifyStatus( CHAR **msgString, Status * status );
static void signalHandler( int sig );

static void *wrapperParams;
static CHAR *exceptionMessage;
static jmp_buf jump;

INT4 initFilters( INT4 argc, CHAR* argv[], CHAR** initMessage )
{
INT4 code;
/* initMessage must come in as a pointer to NULL */
if ( !initMessage || *initMessage ) return Error;
signal ( SIGABRT, signalHandler );
signal ( SIGSEGV, signalHandler );
if ( setjmp( jump ) == 0 ) {

Status status = {0};
CHAR *warning = NULL;
STRVector args;
args.length = argc;
args.data = argv;

initFilters( &status, &wrapperParams, &args, &warning);

code = stringifyStatus( initMessage, &status);
if ( warning && !code ) {
*initMessage = realloc( *initMessage, strlen( warning ) + 1 );
if ( *initMessage ) {
strcpy ( *initMessage, warning );
code = warning;
}
else {
return Error;
}
}
}
else {
```

```
*initMessage = realloc( *initMessage,
strlen( exceptionMessage ) + 1 );
if ( *initMessage ) {
strcpy ( *initMessage, exceptionMessage );
}
return Error;
}
return code;
}

INT4 indexFilters( UINT4* numberIndices, CHAR** indexMessage )
{
INT4 code;
/* indexMessage must come in as a pointer to NULL */
if ( !indexMessage || *indexMessage ) return Error;
signal ( SIGABRT, signalHandler );
signal ( SIGSEGV, signalHandler );
if ( setjmp( jump ) == 0 ) {
Status status = {0};
IndexFiltersParams params;
params.warning = NULL;
params.wrapperParams = wrapperParams;

IndexFilters( &status, numberIndices, &params );

code = stringifyStatus( indexMessage, &status );
if ( params.warning && ! code ) {
*indexMessage = realloc( *indexMessage,
strlen( params.warning ) + 1 );
if ( *indexMessage ) {
strcpy( *indexMessage, params.warning );
code = Warning;
}
else {
return Error;
}
}
}
else {
*indexMessage = realloc( *indexMessage,
strlen( exceptionMessage ) + 1 );
if ( *indexMessage ) {
strcpy( *indexMessage, exceptionMessage );
}
return Error;
}
return code;
}

INT4 conditionData( inPut* data, CHAR** conditionMessage,
MPI_Comm* comm )
```

```
{
INT4 code;
/* conditionMessage must come in as a pointer to NULL */
if ( !conditionMessage || *conditionMessage ) return Error;
signal ( SIGABRT, signalHandler );
signal ( SIGSEGV, signalHandler );
if ( setjmp( jump ) == 0 ) {
Status status = {0};
ConditionDataParams params;
params.warning = NULL;
params.wrapperParams = wrapperParams;
params.comm = comm;
ConditionData( &status, data, &params );
code = stringifyStatus( conditionMessage, &status);
if ( params.warning && ! code ) {
*conditionMessage = realloc( *conditionMessage,
strlen( params.warning ) + 1 );
if ( *indexMessage ) {
strcpy( *conditionMessage, params.warning );
code = Warning;
}
else {
return Error;
}
}
}
else {
*conditionMessage = realloc( *conditionMessage,
strlen( exceptionMessage ) + 1 );
if ( *conditionMessage ) {
strcpy( *conditionMessage, exceptionMessage );
}
return Error;
}
return code;
}

INT4 applyFilters( INT4 beginIndex, INT4 endIndex,
const inPut* data, outPut* result, CHAR** filterMessage,
BOOLEAN finalCall, MPI_Comm* comm )
{
INT4 code;
/* filterMessage must come in as a pointer to NULL */
if ( !filterMessage || *filterMessage ) return Error;
signal ( SIGABRT, signalHandler );
signal ( SIGSEGV, signalHandler );
if ( setjmp( jump ) == 0 ) {
Status status = {0};
IndexFiltersParams params;
params.warning = NULL;
params.wrapperParams = wrapperParams;
```

```
params.beginIndex = beginIndex;
params.endIndex = endIndex;
params.finalCall = finalCall;
params.comm = comm;

ApplyFilters( &status, results, data, &params );

code = stringifyStatus( filterMessage, &status);
if ( params.warning && ! code ) {
*filterMessage = realloc( *filterMessage,
strlen( params.warning ) + 1 );
if ( *filterMessage ) {
strcpy( *filterMessage, params.warning );
code = Warning;
}
else {
return Error;
}
}
}
else {
*filterMessage = realloc( *filterMessage,
strlen( exceptionMessage ) + 1 );
if ( *filterMessage ) {
strcpy( *filterMessage, exceptionMessage );
}
return Error;
}
return code;
}
}

INT4 freeFilters( CHAR** freeMessage )
{
INT4 code;
/* freeMessage must come in as a pointer to NULL */
if ( !freeMessage || *freeMessage ) return Error;
signal ( SIGABRT, signalHandler );
signal ( SIGSEGV, signalHandler );
if ( setjmp( jump ) == 0 ) {

Status status = {0};
CHAR *warning = NULL;

FreeFilters( &status, &wrapperParams, &warning );

code = stringifyStatus( freeMessage, &status);
if ( warning && ! code ) {
*freeMessage = realloc( *freeMessage,
strlen( warning ) + 1 );
if ( *freeMessage ) {
```

```
strcpy( *freeMessage, warning );
code = Warning;
}
else {
return Error;
}
}
LALCheckMemoryLeaks();
}
else {
*freeMessage = realloc( *freeMessage,
strlen( exceptionMessage ) + 1 );
if ( *freeMessage ) {
strcpy( *freeMessage, exceptionMessage );
}
return Error;
}
return code;
}

static void signalHandler( int sig )
{
switch( sig )
{
case SIGABRT:
exceptionMessage = "signalHandler: Caught SIGABRT" FILELINEID;
break;
case SIGSEGV:
exceptionMessage = "signalHandler: Caught SIGSEGV" FILELINEID;
break;
case default:
exceptionMessage = "signalHandler: Caught unknown signal" \
FILELINEID;
break;

}
longjmp( jump, sig );
}

static INT4 stringifyStatus( CHAR **msgString, Status *status )
{
enum { MaxNumLevels = 1024 };
enum { MinMsgStringSize = 1 };
enum { TmpStringSize = 1024 };
CHAR tmpString[TmpStringSize];
size_t msgStringSize = 0;
Status *ptr = status;
INT4 code = status->statusCode ? Error : Nominal;
unsigned level = 0;
if ( !msgString || *msgString ) return Error;
if ( !status ) {
```

```
const CHAR err[] = "stringifyStatus: null status structure" \
FILELINEID;
*msgString = realloc( *msgString, sizeof( err ) );
if ( *msgString ) strcpy( *msgString, err );
return Error;
}
msgStringSize = MinMsgStringSize;
*msgString = malloc( msgStringSize );
if ( !*msgString ) return Error;
memset( *msgString, 0, msgStringSize );
while ( ptr ) {
Status *next = ptr->statusPtr;
size_t totChar = 0;
INT4 numChar;
if ( ++level > MaxNumLevels ) {
const CHAR err[] = "stringifyStatus: too many levels in status
structure" FILELINEID;
free( *msgString );
*msgString = malloc( sizeof( err ) );
if ( *msgString ) strcpy( *msgString, err );
return Error;
numChar = sprintf( tmpString, "\nLevel %i: %s\n", ptr->level,
ptr->Id );
if (numChar < 0 ) {
const CHAR err[] = "stringifyStatus: an error in sprintf()" \
FILELINEID;
free( *msgString );
*msgString = malloc( sizeof( err ) );
if ( *msgString ) strcpy( *msgString, err );
return Error;
}
totChar += numChar;
if (ptr->statusCode ) {
numChar = sprintf (tmpString + totChar, "\tStatus code %i: %s\n",
ptr->statusCode, ptr->statusDescription );
if ( numChar < 0 ) {
const CHAR err[] = "stringifyStatus: an error in sprintf()" \
FILELINEID;
free( *msgString );
*msgString = malloc( sizeof( err ) );
if ( *msgString ) strcpy( *msgString, err);
return Error;
}
totChar += numChar;
}
else
{
numChar = sprintf( tmpString + totChar, "\tStatus Code 0: \
Nomimal\n" );
if ( numChar < 0 ) {
const CHAR err[] = "stringifyStatus: an error in sprintf()" \
```

```
FILELINEID;
free( *msgString );
*msgString = malloc( sizeof( err ) );
if ( *msgString ) strcpy( *msgString, err);
return Error;
}
totChar += numChar;
}
numChar = sprintf( tmpString + totChar, "\tfunction %s, file %s,
line %i\n", ptr->function, ptr->file, ptr-line );
if ( numChar < 0 ) {
const CHAR err[] = "stringifyStatus: an error in sprintf()" \
FILELINEID;
free( *msgString );
*msgString = malloc( sizeof( err ) );
if ( *msgString ) strcpy( *msgString, err);
return Error;
}
totChar += numChar;
++totChar;
if ( totChar > sizeof( tmpString ) ) {
const CHAR err[] = "stringifyStatu: have written beyond bounds \
of string" FILELINEID;
free( *msgString );
*msgString = malloc( sizeof( err ) );
if ( *msgString ) strcpy( *msgString, err);
return Error;
}
CHAR *tmp = realloc( *msgString, msgStringSize += totChar );
if ( tmp ) {
*msgString = tmp;
}
else
{
const CHAR err[] - "stringifyStatus: couldn't allocate memory \
for message string" FILELINEID;
*msgString = realloc( *msgString, sizeof( err ) );
if ( *msgString ) strcpy( *msgString, err );
return Error;
}
}
if ( ptr != status ) LALFree( ptr );
ptr = next;
}
return code;
}

#undef STRINGIFY_HELPER
#undef STRINGIFY
#undef FILELINEID
```

The associated header file, LALwrapperInterface.h will have the following content:

```
#ifndef _LALWRAPPERINTERFACE_H
#define _LALWRAPPERINTERFACE_H

#include "wrapperInterfaceDatatypes.h"
#include "LALDatatypes.h"

#ifdef __cplusplus
extern "C" {
#endif
NRCSID( LALWRAPPERINTERFACEH, "Id: LALWrapperInterface.h$" );

typedef inPut InPut;
typedef outPut OutPut;

typedef struct tagSTRVector {
UINT4 length;
CHAR ** data;
} STRVector;

typedef struct tagIndexFiltersParams
{
CHAR *warning;
void *wrapperParams;
} IndexFiltersParams;

typedef struct tagConditionDataParams
{
CHAR *warning;
void *wrapperParams;
MPI_Comm *comm;
} ConditionDataParams;

typedef struct tagApplyFiltersParams {
CHAR *warning;
void *wrapperParams;
UINT4 beginIndex;
UINT4 endIndex;
BOOLEAN finalCall;
MPI_Comm *comm;
} ApplyFiltersParams;

void initFilters( Status *status, void **params,
STRVector *args);

void IndexFilters( Status *status, UINT4 *numFilters,
IndexFiltersParams *params);
```

```
void ConditionData( Status *status, InPut *inout,
ConditionDataParams *params, MPI_Comm *comm );

void ApplyFilters( Status *status, OutPut *output,
const Input *input, ApplyFiltersParams *params );

void FreeFilters(Status *status, void **params, CHAR **warning );

#ifdef __cplusplus
}
#endif

#endif
```

The C source file and associate header above need only be written once for all dynamically loaded shared object libraries used by the wrapper-API. However, it will be necessary to link this source codes object module into each dynamically loaded shared object.

The LAL algorithms associated with a particular search strategy used in a particular dynamically loaded shared object library are written into the six functions {**`initFilters(), indexFilters(), conditionData(), applyFilters(), freeOutput(), freeFilters()`**} found in the **`LALwrapperInterface.c`** file. The contents of these functions will vary with each search strategy initiated on the LDAS parallel compute cluster using the wrapperAPI as an interface.

VII. WrapperAPI Flow Control

A. Pseudo-Code Illustration of flow control:

1. The following pseudo-code *illustrates* the steps taken by the wrapperAPI. The details are left for the implementation.

```
// wrapperAPI pseudo-code
extern "C" { #include "wrapperInterface.h" }
{ // On every node in Initial Comm perform:
parseCommandLineOptions();
loadDynamicSharedObjects();
errorTestInit( initFilters( argc, argv, initMessage ) );
errorTestIndex( indexFilters( numberIndices, indexMessage ) );
errorTestDC( conditionData( data, conditionMessage, LBComm ) );
MplusLBComm = createWrapperApiLoadBalanceComumunicator();
LBComm = createLoadBalanceCommunicator();
}
while ( notFinished() ) {
if (inLBNode) {
errorTestTF( applyFilters( beginIndex, endIndex,
data, result, filterMessage, finalCall, LBComm ) );
errorTestFree( freeOutput( output, freeMessage ) );
```

```
slaveNodeSendResults();
}
{ // run on master node only:
masterNodeGathersResults();
if (doLoadBalance) masterNodeCalculatesLoad();
masterNodeInformsLDASmpiAPI();
masterSendResultsToResultAPI();
}
// On every node in MPI_COMM_WORLD perform:
MplusLBComm = createWrapperApiLoadBalanceComumunicator();
LBComm = createLoadBalanceCommunicator();
} /* end of while loop */
{ // On every node in MPI_COMM_WORLD perform:
errorTestFree( freeFilters( freeMessage ) );
MPI_Finish();
}
```