Final Report on Phase I activities:

In the Phase I project we concentrated on three technical objectives to demonstrate the feasibility of the Phase II project: (1) the development of a parallel MDSplus data handler, (2) the parallelization of existing fusion data analysis packages, and (3) the development of techniques to automatically generate parallelized code using pre-compiler directives. We summarize the results of the Phase I research for each of these objectives below. We also describe below additional accomplishments related to the development of the TaskDL and mpiDL parallelization packages.

Developed parallel MDSplus data handler

DIII-D researchers observed that when more than one computer tries to access the same node in a MDSplus tree, a slowdown occurs, perhaps because of file locking of the tree. They also observe a slowdown when many computers try to connect to the MDSplus tree, independent of which nodes are being accessed. Figure 1 illustrates this situation. The reason for this bottleneck is not yet understood, but as part of the Phase I of this project, we demonstrated that by using a proxy server one can improve this situation.

We developed a parallel MDSplus data handler for the Phase I project which uses a combination of MPI function calls wrapped into IDL (mpiDL) and function name overloading to set up a proxy server. Figure 2 shows how this new configuration is different from the previous. The proxy server is the only machine in the mpi group to make a connection to the actual MDSplus data server. All other processors request data from the proxy server, thereby unburdening the MDSplus server from having to maintain many simultaneous connections which can cause significant server slowdown.

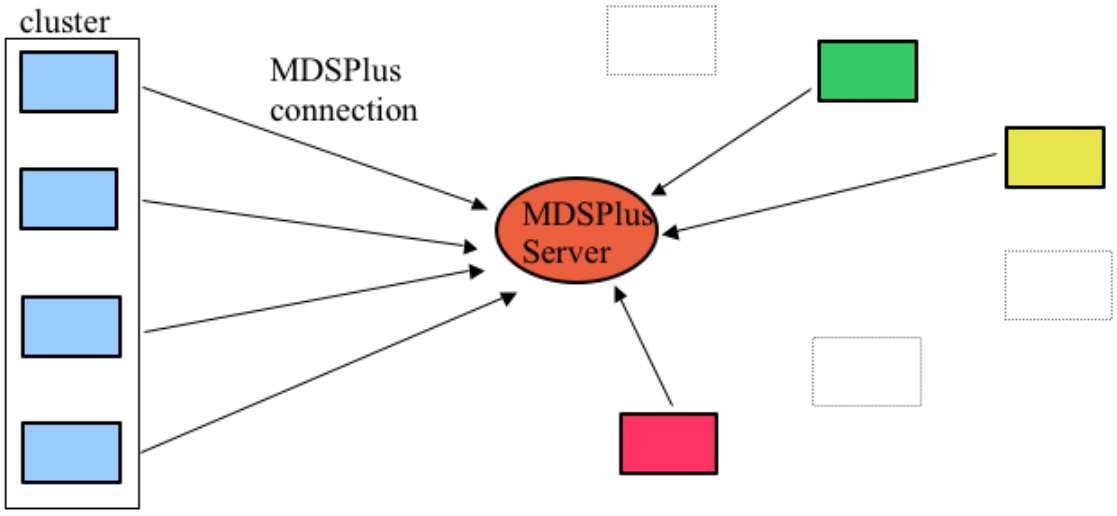

Figure 1: Illustration of the connections made from a cluster to an MDSplus server without the proxy server. In this diagram, researchers not using the cluster are denoted by the boxes on the right (pink, green, and yellow). Researchers at GA notice that some scientists are unable to connect (represented by the grayed out boxes on the right) when a cluster is making many simultaneous connections.

Below is a section of IDL code from the MPI/MDSplus proxy server program. All processors run the same code, but their behavior depends on the unique rank which is assigned by MPI.

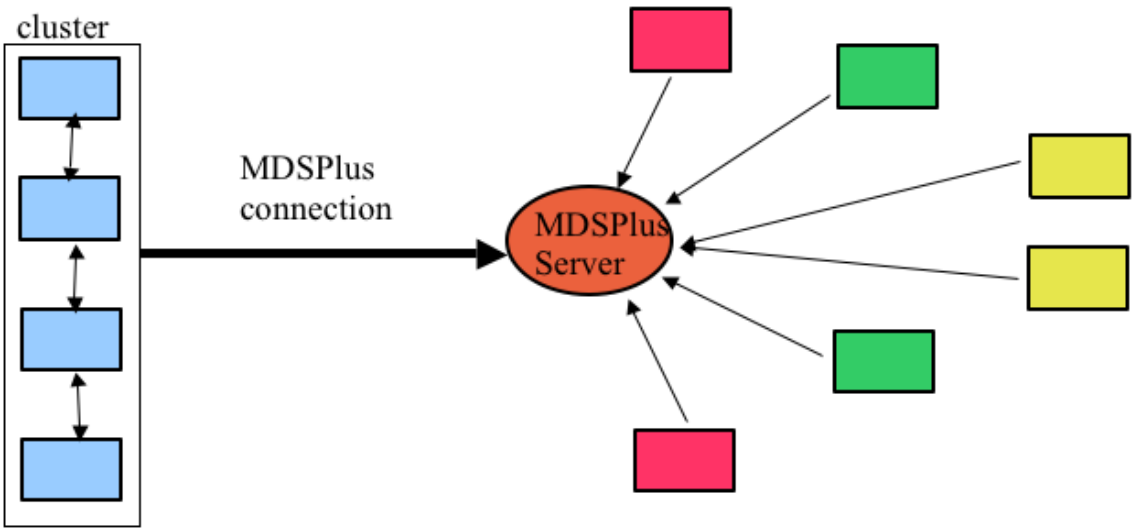

Figure 2: Illustration of the connections made using the MPI/MDSplus proxy server developed as part of the Phase I. By having a head node of the cluster make a single MDSplus connection and distribute the data, the bottleneck is relieved, and more other researchers are able to connect.

The root processor is the proxy server, and all other processors are the slaves. All processors first restore the needed analysis routines. The proxy server restores both the original MDSplus routines and the proxy server routines. The other processors restore the proxy client routines, which have the same function names and data signatures as the original MDSplus routines. This method means that the analysis code can be run using the MPI/MDSplus proxy client/server without modification.

```
IF (myrank EQ root) THEN BEGIN
   RESTORE,filename=compiledfile
   PRINT,'restoring original mdsvalue'
   RESTORE,'/scr_storage1/veitzer/mdsplus/zipfitfile/mdsvalue.orig.sav'
   RESTORE,'/scr_storage1/veitzer/mdsplus/zipfitfile/txmdsproxy.sav'
ENDIF ELSE BEGIN
   MPI_BARRIER
   RESTORE,filename=compiledfile
   RESTORE,'/scr_storage1/veitzer/mdsplus/zipfitfile/txmdsvalue.sav'
   RESTORE,'/scr_storage1/veitzer/mdsplus/zipfitfile/txmdsopen.sav'
   RESTORE,'/scr_storage1/veitzer/mdsplus/zipfitfile/txmdsclose.sav'
ENDELSE
```

The implementation of this parallel MDSplus data handler was tested by retrieving CERQUICK data from an MDSplus server. We show in figure 3 timing information comparing time to retrieve the data both by making straight MDSplus calls (the situation in figure 1) and by using the proxy server (the situation in figure 2) in figure 3. One can see in the figure the server bottleneck as a function of increasing number of processors trying to simultaneously access the data (pink points). Using the MPI/MDSplus proxy server to distribute the data shows an improvement over making direct MDSplus calls (blue points), although a bottleneck is still seen because the underlying MPI communication protocols used in this case were blocking. The red points in figure 3 show the mean time to make a MDSplus call for the proxy server. This shows the proxy server is efficient, because the time to retrieve data from the MDSplus server does not increase with the number of processors. Also, when the proxy server is used, there is only one connection to the MDSplus server made, which allows other researchers to also access the MDSplus server in a timely fashion. These tests were done on the Tech-X Athlon cluster with up to 15 working processors. We also tested the proxy server on a GA cluster with similar results.

Demonstrate parallelization of fusion data analysis codes

Central to doing real-time fusion data analysis is the ability to do a number of basic data analysis functions quickly. For instance, FFTs are basic parts of many fusion data analysis routines. In the case that the FFTs are done on data that is independent, for instance on time-series' which are taken at different spatial locations in the tokamak, a task-based scheme for parallelization can be employed. In the Phase I project we demonstrated that using a task-based parallelization was an effective way to gain speedup in the calculation of toroidal mode power spectra at the NSTX tokamak. A main advantage to this work is that parallelization of the existing IDL code requires no knowledge of parallelization techniques, only knowledge of IDL syntax and language elements.

The data analysis path for calculating mode power involves the following steps. There are twelve Mirnov channels spread out around the tokamak in the toroidal direction. During each shot each channel measures magnetic fluctuation data at that location. During the analysis of the data, a FFT of each channel is made to measure the power spectrum at that location. Then for each frequency, a thresholding method is used to determine if a toroidal mode at that frequency is present. The method of determining if a mode is present requires data from each FFT. The number of modes that can be measured is limited by aliasing in the toroidal direction.

Typically this data analysis is slow for a couple of reasons. The time to retrieve the Mirnov

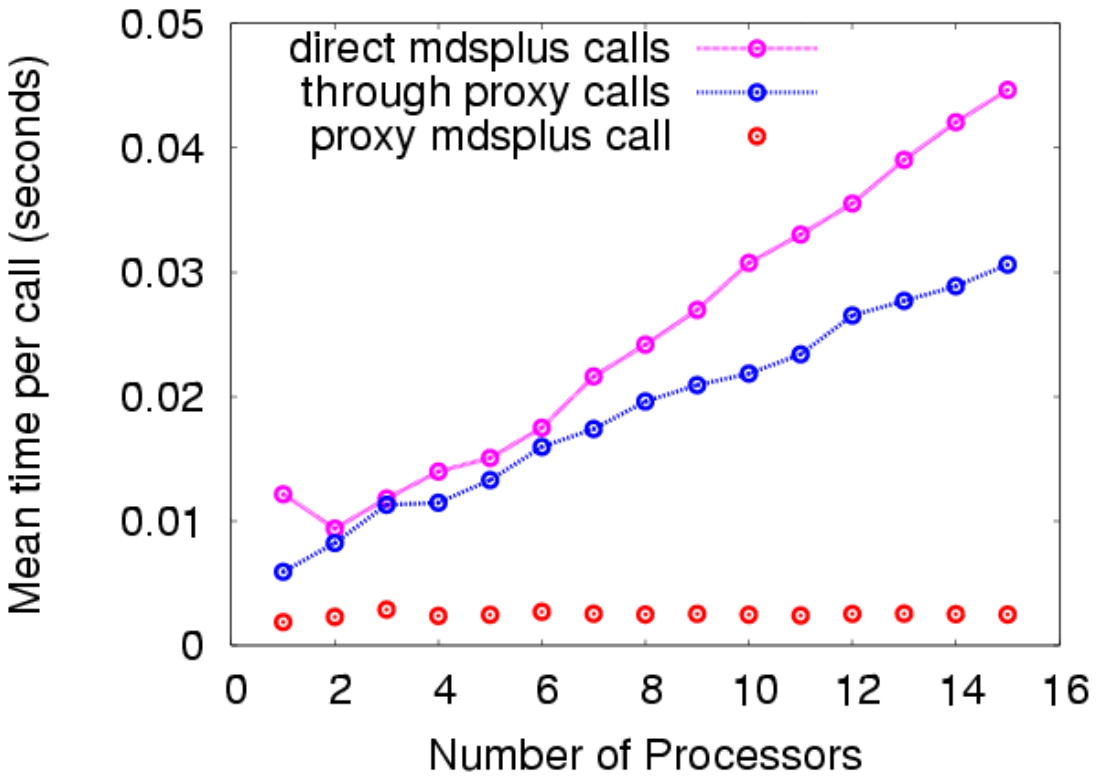

Figure 3: Comparison of mean time to retrieve data from a MDSplus server using direct MDSplus calls versus using the MPI/MDSplus proxy server. The pink points show MDSplus server slowdown as more and more processors attempt to simultaneously access data. The blue points show an improvement but also indicate blocking in the message passing from MPI/MDSplus server to client communication. The red points measure the time for the proxy server to access the MDSplus server.

Coil data from the MDSplus server is reasonable, typically taking about 30-45 seconds to get all of the data from the server. However a typical FFT of the data with a time interval window of .001 seconds and a timestep of .0005 seconds means that the total FFT data over all 12 channels can be so large that the FFT computations are very slow. In addition, the data structure which is created to hold the FFT data of all channels is a 3-Dimensional array of doubles, typically something like [2000,2000,12], and operations on these large arrays such as sorting, searching, and comparison of slices and strides can be exceedingly slow.

In the Phase I project we developed and employed a task farming system for doing the analysis of mode spectra. An example of the result of this analysis can be seen in figure 4. The task farm, called TaskDL, is described in more detail below. The underlying principle is to decompose the problem into independent parts, and to speed up the analysis by using a cluster of computers to simultaneously work on each independent part.

We first decomposed this analysis by doing the FFT analysis part of the problem independently for each Mirnov coil. Each Mirnov channel is independent, so this method was able to achieve very good speedup for the FFT calculation. However the mode number calculation requires information from all FFTs across location, and this caused problems because the amount of FFT and MHD data is large, translating into longer analysis times.

A second, more scalable decomposition of the problem is to have each processor independently

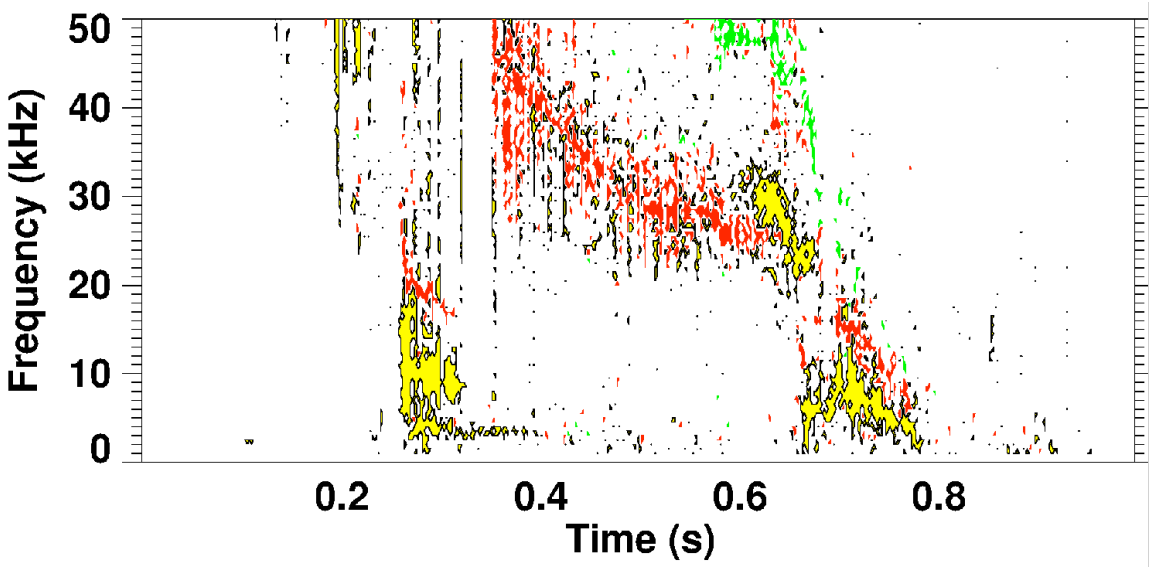

Figure 4: The frequency as a function of time for NSTX shot 109063. The different colors represent different toroidal mode numbers. This analysis requires many tens of thousands of FFTs for each shot of NSTX. This figure is courtesy of J. Menard of PPPL.

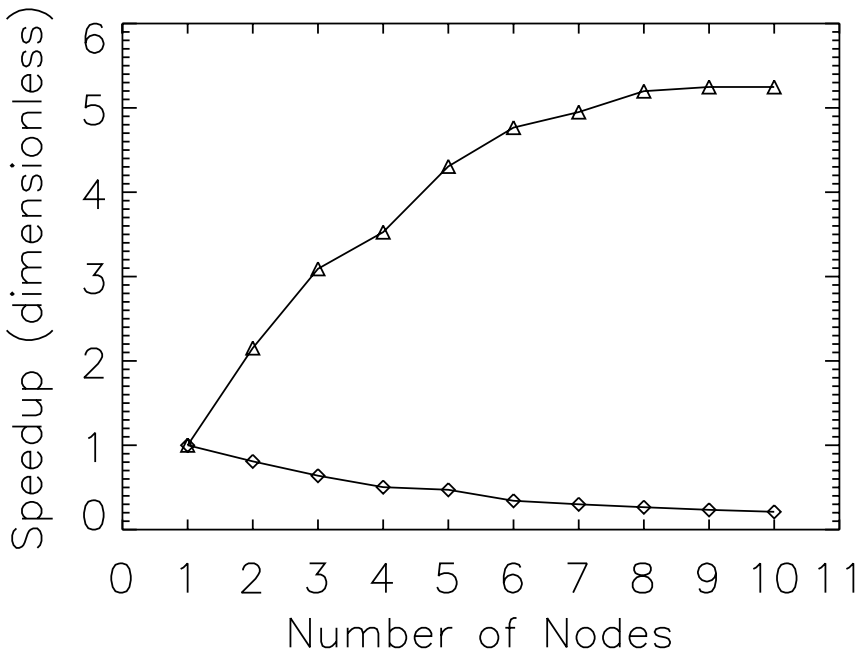

Figure 5: Speedup for toroidal mode spectrum analysis at NSTX achieved using task farming tools developed in the Phase I of this work. Triangles show the speedup for the total amount of time required for the analysis. Diamonds show the MDSplus server slowdown as the number of simultaneous connections increases.

analyze a portion of the data as decomposed in time. Thus if six processors are available to do the analysis each one would take one-sixth of the Mirnov time series data, and do the full mode spectrum calculation using data from all 12 channels. In the Phase I portion of the project we

implemented this decomposition as well, which was effective and showed good speedup. Figure 5 shows the speedup achieved using this method of parallelization (triangles). Running the analysis in serial took approximately 2,140 seconds, while running the analysis in parallel with 6 processors took approximately 440 seconds, a speedup of about 5x.

The immediate slowdown in this decomposition is that each processor must read in all of the data from the MDSplus server. As we saw above, many simultaneous connections to retrieve data from the same node of an MDSplus tree can cause large slowdowns in the data acquisition. Figure 5 also shows the slowdown in retrieval of MDSplus data for this analysis due to many simultaneous connections. The mean time to retrieve MDSplus data here ranges from about 36 seconds for one processor to 170 seconds for 10 processors.

Despite the slower data retrieval times, this method allows one to tailor the analysis to the number of processors which are available, and in fact enables more detailed FFT and mode spectrum analyses to be done because the total amount of data which can be held in RAM over all working processors is greater than that which can be held on a single processor. Thus the time interval (moving window) for the FFT can be increased and/or the timestep can be decreased to a point where a single processor would not be able to do the analysis because of RAM limitations.

Directive Parsing and automated parallelization

We demonstrated in the Phase I project the feasibility of using directive parsing to automate the parallelization process. Directive parsing is a method of putting comments into code, which can then be parsed by a pre-compiler to automatically generate new, parallelized code. For instance, a user may have a loop they believe could be executed faster if split up between independent processors (a loop over files, each producing an independent output, for example). Using directive parsing, instead of rewriting the code to explicitly parallelize the loop, the user inserts an IDL comment (starting with a semi-colon ;) indicating to the pre-compiler that the loop is to be parallelized if possible. The tools we developed for this task then parse those comments and construct the parallel code automatically.

An advantage of directive parsing is that the same code runs in serial and parallel with no changes. If the code is running serially, no pre-compiler is invoked, the comment is ignored and the code runs as before. However, if the pre-compiler is used, then new code is generated to distribute the code to other parallel processors and recombine the results at the end of the loop, and then continue on with the normal execution of the code. This method by default synchronizes all of the parallel processors and it can be a tricky thing to ensure that this type of parallelization will increase performance. Below is an example of how we implement directive parsing into IDL code.

```
PRO test
   seed = 101
   maxx = 1000
   maxy = 2000
   a = DOUBLE(RANDOMU(seed, maxx, maxy))

   ;HPDL DECOMPOSE a(*, BLOCK)
   ;HPDL INDEPENDENT
   FOR i=0, maxy-1 DO a(*, i) = fft(a(*, i))
```

```
   ;HPDL DECOMPOSE a(BLOCK, *)
   ;HPDL INDEPENDENT
   FOR i=0, maxx-1 DO a(i, *) = fft(a(i, *))

   ;HPDL DECOMPOSE a(*, *)
   ;HPDL SINGLE BEGIN
   TVSCL, a
   ;HPDL SINGLE END
END
```

The directives are IDL comments, starting with the symbol ;. The directive `HPDL DECOMPOSE` breaks up the 2-Dimensional array so that it may be broken up among a number of independent processors. The size of the blocks depends on how many processors are available. The directive `HPDL INDEPENDENT` indicates to the pre-compiler that the iterations of the following loop are independent, and may be split up among working processors in an independent manner. The directive `HPDL SINGLE` indicates the beginning or end of a section of code which should only be processed by the master node. The resulting code, contains a generated prolog which defines the parallel routines which are needed to set up the MPI communications within mpiDL, such as `MPI_COMM_RANK()` and `MPI_COMM_SIZE()`, which are not shown here. When processed by the pre-compiler, the generated code for the parallelized test routine looks like:

```
PRO test
   seed = 101
   maxx = 1000
   maxy = 2000
   a = DOUBLE(RANDOMU(seed, maxx, maxy))

   ;HPDL DECOMPOSE a(*, BLOCK)
   ; *** HPDL directive detected!!  ***
   hpdl_var_decompose, A, [0,1], HPDL_DESC_A
   ;HPDL INDEPENDENT
   ; *** HPDL directive detected!!  ***
   ; FOR i=0, maxy-1 DO a(*, i) = fft(a(*, i))
   hpdl_loop_boundaries, 0, maxy-1, hpdl_varStart, hpdl_varEnd
   FOR i=hpdl_varStart, hpdl_varEnd DO a(*, i) = fft(a(*, i))

   ;HPDL DECOMPOSE a(BLOCK, *)
   ; *** HPDL directive detected!!  ***
   hpdl_var_decompose, A, [1,0], HPDL_DESC_A
   ;HPDL INDEPENDENT
   ; *** HPDL directive detected!!  ***
   ; FOR i=0, maxx-1 DO a(i, *) = fft(a(i, *))
   hpdl_loop_boundaries, 0, maxx-1, hpdl_varStart, hpl_varEnd
   FOR i=hpdl_varStart, hpdl_varEnd DO a(i, *) = fft(a(i, *))

   ;HPDL DECOMPOSE a(*, *)
```

```
   ; *** HPDL directive detected!!  ***
   hpdl_var_decompose, A, [0,0], HPDL_DESC_A

   ;HPDL SINGLE BEGIN
   ; *** HPDL directive detected!!  ***
   IF hpdl_i_am_master() THEN BEGIN
   tvscl, a
   ;HPDL SINGLE END
   END ; END HPDL SINGLE
END
```

Here, the independent loops have been decomposed such that the computation is automatically spread out among many processors. The generated code is much more complex, but because the code is automatically generated, this is not a concern. This example showed automated parallelism can be achieved using directives and demonstrated that serial code can be parallelized automatically and without changing the functionality of the serial code.

Additional Phase I work: Parallelization Methods Developed in Phase I

A major part of the Phase I project was the development of two methods for parallelization of IDL code. The first method, called TaskDL, creates a farm of cluster processors which can independently execute IDL routines. The second method is the incorporation of MPI library routines into the IDL language so that they may be called as native system routines. Both of these methods have played a key role in the commercialization aspects of this project. We briefly describe here the work which was done in the Phase I project on the development of these parallelization packages.

TaskDL:

An integral part of enabling task-based parallelization of the NSTX data analysis path was the development of the TaskDL software package. In addition to providing an integrated interface for researchers at PPPL to parallelize their analysis codes, TaskDL is a major part of the commercialization plans for this project. Using TaskDL to demonstrate the effectiveness of parallelizing NSTX data analysis codes provided important feedback on the development of the TaskDL package.

TaskDL coordinates farms of independent workers which all serve to do independent analyses. TaskDL is written in C and IDL, with underlying C function calls added as system routines to IDL using the Dynamically Loaded Module (DLM) functionality. Needed IDL code is precompiled which allows for workers to be commissioned in IDL's runtime mode, which is both efficient upon startup and is cost effective to license on a cluster of computers.

TaskDL provides a robust way for users to utilize their serial IDL code without major changes. Typically the user will use a programmatic IDL script to create a workspace for the task farm (also called a tuple space), commission workers (typically remote hosts on a cluster) into the tuple space, and add task tickets (tuples) to the space so that workers may process them. Workers then pull tasks from the tuple space as they become available to process a task, and database locking mechanisms are used to determine race conditions and ensure that a given task is only

processed by a single worker. Tasks may also be tagged to a specific processor, so that only that worker will process the task. In fact, this is the method by which workers are decommissioned from the tuple space. They receive a directed task which tells them to shut down IDL and cease processing tasks.

A number of new IDL system routines are exposed to the user, providing the basic programmatic task farm functionality. They are:

- `TASKDL_CWORKSPACE` creates a common workspace or session directory for the task farm. This workspace contains the database which contains the tasks and information about the remote workers. Log files and other information about the session are also put here, and all commissioned workers look here to find tasks.

- `TASKDL_CWORKER` commissions workers into the workspace. This function takes care of setting up and executing IDL on a remote host, and points the remote worker to look at a particular workspace to find tasks and write log files.

- `TASKDL_CTT` is a routine which creates task tickets for the remote workers to process. The task tickets is a valid IDL routine with associated input arguments. The task is placed into a database which is then polled by workers. The routine which is to be executed by the remote worker must be precompiled, so the `TASKDL_CTT` routine allows the user to specify the location of the compiled routine.

Using these functions, researchers can create and manage persistent task farms from within IDL. We have also developed a graphical user interface which allows users to monitor and control existing task farms.

TaskDL workers, once launched on a remote host, will run independently of the computer which started the job. This is achieved by executing a shell script on the remote host over an ssh channel in the background (nohup). The ssh channel is then disconnected and the shell script, which sets certain required environment variables and launches IDL in runtime with the TaskDL control program running, continues to run until the IDL session is completed.

TaskDL workers will continue to poll the tuple space looking for new tasks even if there are currently no tasks to be done. This allows the TaskDL software to be run in daemon mode, so that as new tasks are put into the tuple space they will be processed without needing to restart the whole task farm process. This is useful for analyzing a series of data which is becoming available in real-time, such as shot after shot in a tokamak experiment.

We have also included support for cluster scheduling software, such as PBS, which is important for running TaskDL on clusters which are being shared by many users. The queuing scheduler, openPBS, provides allocated hostnames to a submission script. These hostnames are then used by a specifically designed IDL batch script which converts the hostnames to worker IDs appropriate for TaskDL, and uses these IDs to commission workers on those hosts. Our integration of TaskDL with cluster scheduling software adds to the usefulness of the software, and provides a lightweight solution for IDL researchers who have access to shared clusters but do not want to expend a lot of effort to get their code parallelized.

mpiDL:

When the code to be parallelized does not involve independent tasks it is necessary to incorporate communication between cluster processors. A typical example of this is problems which are parallelized by domain decomposition, where each processor has a portion of the domain, say a

block of a large array, and needs to communicate information across the boundaries of the block to neighboring processors. There are other cases where communication between processors is needed, such as the MPI/MDSplus proxy server described above.

To enable communication between processors, we have implemented a subset of MPI library routines into IDL using the DLM interface, called mpiDL. Tech-X and PPPL have previously developed prototypes for mpiDL, which contained the most basic MPI communication routines. As part of the Phase I project, we added new MPI routines which added to the functionality of mpiDL. These include a suite of one-to-many communication routines such as `MPI_SCATTER`, `MPI_ALLGATHER`, and `MPI_ALLTOALL`. We also added `MPI_BARRIER`, which allows the user to synchronize all processors.

In addition to adding more MPI functionality, we developed some specialized routines based on MPI communication calls which allow the user to pass specific data structures between processors. These routines are not part of the MPI library of functions, but represent added message passing functionality that can be tailored to specific analysis codes. For instance, we developed a specialized send and receive routine which passes both an array of values and an integer which denotes a status flag in a single message. The size of the array is variable, and the array can contain data of any type, including strings. Using only basic MPI routines this communication would require at least four messages, indicating (1) the size of the array to be passed, (2) the type of data in the array, (3) the array itself, and (4) the status.