ORNL/TM-13299

Computer Science and Mathematics Division

Mathematical Sciences Section

CUMULVS USER'S GUIDE
COMPUTATIONAL STEERING AND INTERACTIVE VISUALIZATION IN DISTRIBUTED APPLICATIONS

James Arthur Kohl and Philip M. Papadopoulos

Oak Ridge National Laboratory
P.O. Box 2008, Bldg 6012, MS 6367
Oak Ridge, TN 37831-6367

Date Published: August 1996

Research supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
managed by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC05-96OR22464

Contents

CUMULVS USER'S GUIDE
COMPUTATIONAL STEERING AND INTERACTIVE VISUALIZATION IN DISTRIBUTED APPLICATIONS

James Arthur Kohl and Philip M. Papadopoulos

Abstract

CUMULVS is an infrastructure library that allows a programmer to easily extract data from a running parallel simulation and send the data to a visualization package. CUMULVS includes the capability to steer user-defined parameters in a distributed simulation. An additional feature being added to CUMULVS is the ability to perform user-directed checkpointing and restarts of parallel programs across a heterogeneous network. The interface is kept as simple as possible. The programmer merely defines the decomposition of parallel data and the number and types of scalar parameters. CUMULVS supports the simultaneous viewing of multiple dynamically attached front-end visualization programs. A single interface routine handles the attachment/detachment protocols, the selection and sending of data, and updating steering parameters. CUMULVS allows each viewer to interactively select the granularity and extent of data that it desires to see. This allows the user to effectively visualize large datasets without exhausting the network bandwidth. CUMULVS also supports user-directed checkpointing and restarts across a heterogeneous machine. The visualization and checkpointing features of CUMULVS are independent.

1. Introduction

CUMULVS is an infrastructure library that allows a programmer to add interactive steering and visualization to an existing parallel or serial program. From the application code perspective, the setup is very simple. After describing how data fields are decomposed across processors, and which parameters are adjustable from the front-end, the program make a single subroutine call in the body of the main loop. This call directs when CUMULVS can communicate with and send data to attached viewers. The return status from this subroutine can be queried to determine if the viewer program has adjusted (steered) any parameters. CUMULVS guarantees that all nodes in the parallel program will receive updated parameters at the exact same timestep as long as all nodes call the interface subroutine at each timestep. The subroutine consumes very little overhead when no viewers are attached – the cost of a single message probe.

CUMULVS "understands" standard HPF-style block and block-cyclic decompositions as well as particle decompositions. The system allows a viewer to examine the data in a parallel program as if it were available as a large monolithic array. The software manages all the details of what data to pack and send to a front end and all the details of attaching a viewer to a running program. Currently, CUMULVS uses PVM as its message passing substrate. The essential ingredient in PVM is the allowance for anonymous tasks to communicate with each other without both tasks being started at the same time. CUMULVS can be ported to any other messaging system that allows these dynamics. A name registry is used so that a viewer needs only to know the "name" of the code in order to start communicating with it.

This user guide explains how CUMULVS interprets data decompositions, how viewers interact with a running program and gives manual pages for the supported user interface routines.

2. How to Use CUMULVS

CUMULVS is intended for programmers to easily add real-time visualization and steering to iterative programs. A large number of problems fall into this category making CUMULVS a widely applicable but not universal tool. The pseudo-code in Figure 1 illustrates the typical statement sequence that a programmer would follow to define distributed data fields, steerable parameters, and enable visualization.

The predominant complication is setting up CUMULVS to understand the user's distribution of data so that the software can automatically select subsets as required by an attached front-end. Once this setup is complete, "all the action" occurs in a single subroutine call, `stv_sendToFE()`. The programmer never worries about how a visualization package attaches to a CUMULVS program. Steering parameters are guaranteed to be updated at the same iteration across the entire parallel program as long as the programmer calls `stv_sendToFE()` at each iteration. The following section documents the data

Initialize CUMULVS data structures (`stv_init()`)

Define data decomposition (`stv_decompDefine()`)

Define data field with a previously defined decomposition (`stv_fieldDefine()`)

Define steering parameters (`stv_paramDefine()`)

Start main iterative loop

<usual calculation>
nchanged = `stv_sendToFE()`
<program response to *nchanged* steered parameters>

End of main iterative loop

Figure 1: Typical execution order for a CUMULVS program

decompositions that CUMULVS understands.

3. Data Decompositions

CUMULVS understands regular block decompositions, block-cyclic decompositions a la HPF, particle decompositions, overlapping block decompositions, and a user-defined block decomposition. To define any decomposition, a program must supply:

- The dimension of decomposition (1D, 2D, 3D) (dim)
- The global upper and lower bounds of the data array (glb, gub)
- The dimension of logical processor decomposition ($prank \leq dim$)
- How each axis of the array is decomposed ($axisInfo$)

Data is decomposed onto a logical array of processors. For example, a three-dimensional array might be decomposed onto a two-dimensional array of processors. This means that one axis of the array lies entirely within a single process. Sometimes

3.1. Block Decompositions: stvBlock

The stvBlock data decomposition distributues data in a block manner across logical processors. Blocks of data do not wrap. The number of data elements assigned per processor is based on the formula

$$n = \lceil N/p \rceil$$

where n is the number of elements assigned to a processor, p is the number of processors for this coordinate, N is the total number of elements. CUMULVS will assign n elements to processors until the total number of elements has been exhausted. This formula only poses difficulties when N is not evenly divisible by p. For example, in an extreme case where $N = 11$ and $p = 10$ then the first five processors will be assigned 2 elements each, the sixth processor will be assigned one element, and the remaining will be assigned none. If the user desires a different block-type decomposition, the stvExplicit may have to be used.

3.2. Block-Cyclic Decompositions: stvCyclic

The stvCyclic assigns blocks of data in a cyclic manner to processors. The default cycle size is 1, but is user selectable. Blocks of size c are assigned to processors until the number of processors is exhausted. The assignment wraps and continues starting at logical processor 0. For example, if the number of processors p is 5, the number of elements N is 14 and the cycle size c is 2, then processors $0 - 4$ will be assigned the following array indices:

$proc0$ $\{0, 1, 10, 11\}$

$proc1$ $\{2, 3, 12, 13\}$

$proc2$ $\{4, 5\}$

$proc3$ $\{6, 7\}$

$proc4$ $\{8, 9\}$

Note that that above example uses C 0-based indexing. Fortran 1-based indexing would have each array index increased by one.

3.3. Explicit Decompositions: stvExplicit

If neither of the above decomposition schemes statisfy the user requirements, then and uneven or explicit decomposition may be used. In this scheme the user specifies an lower and upper bound of array indices that held on each processor. Returning to the example in the subsection 3.1, suppose that one desires processor 0 hold the first two array indices and the remainder should hold on index each. Then, lower and upper bounds of the explict decomposition per processor my be specified as.

$$\begin{aligned} lower_i &= i + 1 \text{ if } i \neq 0. \;\; 0 \; otherwise. \\ upper_i &= i + 2 \;\; i \neq 0. \;\; 1 \; otherwise. \end{aligned}$$

The explicit data decomposition should be used only when other decompositions will not work.

3.4. Collapsed or On-processor Decompositions: stvCollapse

If all the indices for a coordinate should reside on a single processor, the decomposition is said to be collapsed. In this case, the user should use the decomposition type stvCollapse.

4. Parameters

CUMULVS supports scalar steering parameters. The programmer declares the type (e.g., float, int) of the parameter, name and a storage location. Front-end viewers. Once the parameter is declared, it will only be changed during a call to `stv_sendToFE`. This routine returns the number of parameters that have been called. A programmer may query to check if a parameter has been changed by callind `stv_isParamChanged`.

5. CUMULVS Viewers

This section describes the construction of front-end "viewer" programs for use with CUMULVS. The CUMULVS package already comes with a standard text-only viewer, a standard AVS-compatible viewer, and a sample custom TCL / TK viewer for a particle-based CUMULVS game called "FLYME." These viewers should be sufficient for typical user needs, and special custom viewers need not be created. However, if a special viewer is desired, these example viewers represent the best source of information for designing a custom user viewer. The provided viewers constitute complete working examples of how viewers interact with user applications using CUMULVS. The code in these viewers is given freely, to be copied and used as needed for the construction of other viewers, as long as the enclosed copyright statements are included.

In addition to the sample viewers, several key viewer routines are documented, in Section 7, with standard man pages. It should be noted that there is presently only a C language interface for CUMULVS viewers, with no Fortran or other language support. It is assumed that the functionality of viewers is best executed in traditional C or C++ programming. While it is certainly possible to invoke the provided viewer routines from other languages, there is no specific support provided for doing this with CUMULVS.

There are several classes of functions provided for use with CUMULVS viewers, to support the collection of data field values, to provide computational steering, and to assist the viewer writer with other viewer utilities. A typical viewer consists of some subset of these viewer library functions. The following subsections describe the viewer library in more detail.

5.1. Data Field Collection

The primary use of CUMULVS is for the interactive collection of data field values from a running application, to support graphical animations or other analyses. There are a number of functions provided in CUMULVS for handling the necessary data field operations in viewers, including initializing communication with the user application, requesting data fields, collecting data frames, allocating data frame storage, and dumping data field values.

All viewers must call `stv_viewer_init()`. This function initializes a link with a specific user application and gathers information about the data fields and parameters that are available from that application. The call to `stv_viewer_init()` returns an `STV_VIEWER` instance which is used in subsequent viewer routine calls to identify the specific application being "viewed" (thereby allowing a single viewer to connect to multiple applications if desired).

Once / if CUMULVS finds the desired application, the viewer may select some number of data fields to be requested for collection. The `stv_clear_view_field_select()` routine resets this selection, and then individual `STV_VIEW_FIELD` elements can be chosen for a data field request. The `STV_VIEW_FIELD` elements are obtained by name using the `stv_get_view_field_name()` routine, and each can then be selected by setting the `STV_VIEW_FIELD−>selected` flag to `stvTrue`. The set of fields selected for a particular field request is considered a "view field group" (see `STV_VIEW_FIELD_GROUP` below).

The viewer can select a specific data type for each data field in a view field group (VFG). The value of `STV_VIEW_FIELD−>view_type` for a given data field can be set to any of the supported CUMULVS data types. This data type can be different than the original type defined for the field in the application. For example, if a simulation used double floating point data for its computation, a viewer could request the data in single precision or even integer format, which might be more suitable for simple graphical presentations. Similarly, the viewer can specify the storage order for each field in a VFG by setting the value of `STV_VIEW_FIELD−>view_storage_order` (`stvColumnMajor` for standard Fortran storage order, or `stvRowMajor` for standard C storage order). The data values will then be rearranged accordingly during collection to support the desired array addressing in the viewer. So, for example, an application written in C language can be viewed more naturally using AVS, which assumes a Fortran storage order.

If the data type or storage order are not specified before requesting the view field group, then the data field will be collected and provided to the viewer using the original data type or storage order, as declared in the actual user application. Otherwise, CUMULVS automatically converts the data type and storage order at each task of the application before the data is transferred to the viewer.

For each view field group, the viewer must specify the portion of the computational space which is to be collected for viewing. This area is referred to as the "visualization region," or "vis region" for short. The vis region consists of a set of upper and lower coordinate bounds for each axis of the computational domain, as well as a "cell size" for each dimension. The cell size indicates the granularity of data values which are to be returned. For example, a cell size of "2" for the "X" axis corresponds to collecting every other data value along that axis. Then combining this with a cell size of "3" for the "Y" axis would result in collecting 1 out of every 6 data values. So a complete vis region specification might include the set of data values between 10 and 50 along the "X" axis with a cell size of 5 (every fifth data value - 10, 15, 20...), and those between 30 and 40 along the "Y" axis with a cell size of 2 (every other data value - 30, 32, 34...), thus resulting in 54 total data values (including data points addressed as (10,30), (10,32),...; (15,30), (15,32)...; etc). The vis region bounds and cell sizes for each axis then determine precisely which data points will be collected for each viewer "data frame."

Note that a single vis region specifies the collection area for the entire group of data fields in a VFG. If different vis regions are desired for different data fields they must be requested in separate VFGs. A given data field can, however, occur repeatedly in any number of view field groups.

Once the desired data fields have been selected and the vis region has been specified, a field request is sent to the application tasks using the `stv_viewer_request_field()` routine. This routine returns an `STV_VIEW_FIELD_GROUP` instance that represents the group of data fields requested. The `STV_VIEW_FIELD_GROUP` is used in other viewer routines to manipulate aspects of the incoming data frames, as well as to terminate or release the interactive data field connection with the application.

Before a data frame can be received by a viewer, the appropriate storage must be allocated. The viewer must calculate the total storage necessary for each field in a data frame and save this information in a special array. This is accomplished automatically using the `stv_allocate_Field_arrayDecl()` routine. If the actual data frame storage is to be allocated by CUMULVS (as is done with the sample text-only viewer), then the `stv_allocate_Field_Data()` routine is called for each field in the group. This routine is used only for those fields in the VFG with regular contiguous data decompositions. For fields that are particle fields, the space for each particle is automatically allocated for the viewer by CUMULVS. Afetr a data frame has been used by the viewer, it can be freed using the `stv_free_field_array()` routine. Any particles allocated can be freed by a call to the `stv_reset_Field_Particles()` routine.

If a viewer needs to use externally allocated memory for storing its data frames (as is true with the AVS viewer), then the viewer need only set the proper pointer to that memory in the VFG structure. Specifically, `VFG−>field_values[ field_index ]` must be assigned the address of the pre-allocated memory, where `field_index` is the index of the particular data field in the group list. It should be noted that this VFG pointer setting must be cleared (back to `(STV_VALUE) NULL`) before recycling the VFG with a new `stv_viewer_init()` call. Otherwise, CUMULVS will assume that it had allocated that memory and will attempt to free it.

To actually receive a data frame, the viewer calls the `stv_viewer_receive_frame()` routine. This routine returns the VFG of the received frame (in case there are several VFGs), a `restart` flag, and a return status code. There are several return status values depending on the outcome of the data frame collection with the application. If an `stvStatusOk` is returned, then the VFG argument contains a handle to the view field group that has collected a complete data frame. If a `stvStatusBadFrame` is returned it means that all tasks in the application have sent their data, but some were not completely up-to-date with the last requested vis region resulting in an inconsistent data frame. In this case the data frame can simply be discarded. Otherwise, something catastrophic has happened and the viewer should disconnect from the application. If the `restart` flag has been set (to `stvTrue`) then the application has merely reconfigured, and the viewer should disconnect and try to re-attach to the same data fields.

Once a complete data frame has been received, the viewer needs to send the application an "XON" to release it for the next iteration. This is done using the `stv_viewer_send_XON()` routine. The sooner the XON is sent, the less intrusion and overhead is expended by the application in waiting for it. I.e. it is a good idea to send the XON immediately, before processing the new data frame. If the iteration time for the application is sufficiently large (and the size of the data frame is not too immense), the application might not wait for the XON at all. The XON could already be there waiting for the application when it polls for permission to continue with the next iteration.

The vis region for a VFG can be modified on-the-fly by a viewer using the `stv_viewer_set_VisRegion()` routine. This routine records the new set of region bounds and cell sizes and sends the application an update message with the new vis region. CUMULVS takes care of verifying that the next data frame is collected using the proper vis region, and will return a `stvStatusBadFrame` return code if any of the application tasks did not receive the update in time. The frequency of data frames (counted in number of application iterations between frames) can be modified using the `stv_viewer_set_VisFrequency()` routine. CUMULVS insures that the relative timing between the application and the viewer is maintained, to support the loose synchronization required for computational steering.

When a data frame is collected, CUMULVS can be set to automatically dump out a textual representation of the data it has received. This is done in the viewer using the `stv_setopt()` routine with either of the `stvOptDump` or `stvOptDumpAll` options. The `stvOptDump` option dumps only the final data frame, where the `stvOptDumpAll` option dumps all intermediate data frame states, as they occur after each individual application task sends its piece of the data frame. If the viewer needs to otherwise dump the data field values, then the `stv_dump_field_array()` routine (or one of its variants, `stv_dump_vfg_field_array()` for viewers, or `stv_dump_user_field_array()` for user applications) can be called to display any complete data frame.

5.2. Steering Computations

Aside from collecting data frames from running applications, CUMULVS viewers can also remotely modify an application's computational parameters on-the-fly. This process is known as "computational steering." Often this is a useful capability when the user desires to experiment with various parameters in a computation. Or perhaps viewing the intermediate results of a computation can reveal a problem or a new opportunity to manipulate the application. Such interactive control can save countless hours of wasted computation time waiting for final application results that might have begun experiencing problems in the first few iterations.

Before a viewer can initiate steering with a user application it must invoke the `stv_viewer_steering_init()` routine. This routine performs the equivalent of a special data field request, creating a loosely synchronized connection with the user application. The viewer uses the connection to transfer updated steering parameters to the application. The loose synchronization guarantees that all tasks in the application will apply those updates at the same "time," or point in the computation. Note that because this steering connection utilizes a type of field request, the viewer must process the incoming data field protocol using repeated calls to `stv_viewer_receive_frame()`. This routine automatically maintains the steering connection by returning XONs to the application when all tasks have sent their acknowledgements for a given iteration. Failure to call `stv_viewer_receive_frame()` while steering will result in the application hanging (waiting for XONs). The viewer should not explicitly send a steering XON.

Once steering has been successfully initialized, a specific steering parameter can be controlled by acquiring the appropriate steering token. The parameters can be looked up by name using `stv_get_view_param_name()`, which returns a `STV_VIEW_PARAM` instance. The steering token for a particular parameter is then obtained using the `stv_viewer_steering_request()` routine. If the token for a parameter is not already in a viewer's possession, then the `STV_VIEW_PARAM->token` value will be set to `stvSteerToken` upon return from the request call.

If, however, the token is already in use, then the value of `token` will either be `stvSteerRqstd` or `stvSteerNot`. The `stvSteerRqstd` value means that the steering request was successfully submitted but the token is unavailable. In this case, when the viewer which currently has the token releases it, CUMULVS will broadcast a message informing all the requesting viewers. So subsequent calls to `stv_viewer_steering_request()` merely check for that release message and, if found, attempt again to acquire the steering token. Note that calling `stv_viewer_steering_request()` with the `block` flag set to `stvTrue` can result in deadlock if the token does not become available. This is due to the application starving for steering connection XONs. If the viewer is not assured of obtaining a given steering token, then, while waiting for a steering token, a viewer should alternate between a non-blocking call to `stv_viewer_steering_request()` and a call tp `stv_viewer_receive_frame()`.

If the value of `STV_VIEW_PARAM->token` is `stvSteerNot` after a call to `stv_viewer_steering_request()`, then some error has occurred, such as an internal system error or an incorrect invocation parameter. In this case, the steering parameter will not likely be obtained, and the viewer should disconnect from the application using `stv_viewer_send_FieldHaltGroup()`, `stv_viewer_send_FieldHaltAll()` or by exiting. For steering-only viewers that do not have other view field groups for data field collection, the external symbol `extern STV_VIEW_FIELD_GROUP STV_STEER_VFG` can be used in either of the `FieldHalt` library calls.

To actually set the value of a steering parameter for which the steering token has been obtained, the viewer can call either `stv_viewer_steer_parameter()` for scalar parameters or `stv_viewer_steer_vparameter()` for vector parameters. These routines copy the viewer data, in the form of an `STV_VALUE` data value pointer, over into the viewer parameter structure, and then sets the `changed` flag for that parameter. When all steering parameter values have been set as desired, the `stv_viewer_send_NewParams()` routine is called to pass the new parameter values to the application tasks. This routine checks the `changed` flags for each parameter and updates only those parameters with new values.

When all changes to a steering parameter have been completed, a viewer can release the steering token with a call to `stv_viewer_steering_release()`. This call will relinquish the steering token and, as stated above, will broadcast a message to any other viewers that have requested the given steering parameter and are waiting for the token. If a viewer exits without releasing control of a steering parameter, the token should be automatically freed by CUMULVS.

Aside from traditional scalar and vector computational steering parameters, CUMULVS also supports a special type of steering parameter known as an "indexed" parameter. For certain kinds of simulations, especially particle-based applications, there may be many replicated objects or entities to be steered. If it is necessary to manipulate individual instances of these objects, or if the number of instances in the application can grow or shrink, then indexed steering parameters are essential. Using indexed parameters, only one set of steering parameters are defined for a single object instance. Then in addition to the regular application parameters, one additional "index" parameter is defined. When a set of steering parameters is passed to the application, the index value is extracted first to determine which object instance is to be steered, and then the remaining parameter values are applied only to that one instance.

The special index parameter is defined by including the `stvIndex` flag with the regular parameter data type in the call to `stv_paramDefine()` (or `stv_vparamDefine()`, as in `stvInt | stvIndex`. The index parameter can be of any legal CUMULVS data type, and its value is not interpreted internally by CUMULVS. It is left to the application to properly utilize the custom index value in referencing its object instances.

It should be noted that the process of acquiring steering tokens still applies to indexed steering parameters. In fact, steering tokens are granted for each desired value of a particular steering index, so that different instances from the same object set can be simultaneously steered. For example, if one viewer wishes to steer an object instance "A", and another viewer wishes to steer a different instance "B" from the same object set, then each viewer will obtain their own "indexed token." Note that steering tokens are not generated until they are requested, so CUMULVS need not know the entire range of possible index values, nor allocate them all, to properly coordinate the tokens.

5.3. Other Viewer Utility Functions

6. User Library Manual Pages

stvfinit() stv_init()

Initialize CUMULVS data structures and declare the name that viewer front-ends will look for to communicate with the application

Synopsis

C `int info = stv_init( char *appname, int msgtag, int nproc, int nodeid )`

FORTRAN `stvfinit( appname, msgtag, nproc, nodeid )`

Parameters

`appname`	–	name of the application. Used by front-end viewers to look up and connect to the running CUMULVS application. This name does not have to be the same as the executable name.
`msgtag`	–	message tag for CUMULVS to use for communication with the application. The user's code must not receive an messages with type msgtag.
`nproc`	–	number of processors that make up the entire application.
`nodeid`	–	node number of the calling process. Must be in the range $[0, nproc - 1]$
`info`	–	return status parameter

Discussion

The routine `stv_init` initializes CUMULVS data structures and declares the size and name of the user application. All tasks must call `stv_init` once. The call is not collective, but viewing front-ends will not be able to communicate with the application until all tasks have called this routine.

CUMULVS uses a single message tag to communicate with the program. All tasks must use the same user-specified tag. A tag is needed for messaging systems that do not implement context (like PVM). Unpredictable behavior will occur if the user's code receives messages intended for the CUMULVS library. Most notably, complete wild-card receives (e.g., `pvm_recv(-1,-1)` will cause programs to fail. "Semi-wildcard" receives (e.g., `pvm_recv(-1,tid)` or `pvm_recv(tag,-1)`

CUMULVS needs to know how many processes or tasks make up a parallel application. Each task must have a unique node id that is in the range of 0 to $nproc - 1$.

Examples

```
C:
   info = stv_init( "csimple", 1000, nproc, i );
```

FORTRAN:

```
call stvfinit( 'csimple', 1000, nproc, i, info )
```

Errors

Error conditions that can be returned by `stv_init`.

Name	Possible cause
stvStatusBadParam	Could not store application name for lookup

See Also

`stv_fieldDefine`, `stv_vParamDefine`, `stv_isParamChanged`, `stv_sendToFE`

stvfdecompdefine() stv_decompDefine()

Defines a data decomposition. CUMULVS needs to know the parameters of decomposition to know how to pack data for a front-end viewer

Synopsis

C
```
int decompId = stv_decompDefine( int dim, int *axisType,
                                 int *axisInfo, int *axisInfo2,
                                 int *glb, int *gub,
                                 int procRank, int *procShape )
```

FORTRAN
```
stvfdecompdefine(  dim, axisType, axisInfo
                   axisInfo2, glb, gub,
                   procRank, procShape, decompId )
```

Parameters

- `dim` – integer specifying the dimension of the distributed array
- `axisType` – integer array specify the decomposition type of a particular dimension of an array. Valid values are: *stvBlock*, *stvCyclic*, *stvCollapse*, and *stvExplicit*.
- `axisInfo` – integer array that qualifies `axisType`. Entry i in `axisInfo` corresponds to entry i in `axisType`.
- `axisInfo2` – integer array that qualifies `axisType`. Entry i in `axisInfo2` corresponds to entry i in `axisType`. Only referenced if a decomposition type is set to *stvExplicit*.
- `glb` – integer array that specifies the global lower bounds of the decomposed field.
- `gub` – integer array that specifies the global upper bounds of the decomposed field.
- `procRank` – integer the specifies the rank of the logical processor array.
- `procShape` – integer array the specifies the extent of the logical processor array in each dimension.
- `decompId` – integer identifier for this decomposition. Used in `stv_fieldDefine`.

Discussion

The routine `stv_decompDefine` defines all parameters needed to describe a decomposition to CUMULVS. Axes of an array may be decomposed in different ways on the different axes. For example, a three-dimensional array might be decomposed in a block manner in the first dimension, cyclic in the second dimension, and on-processor (or collapsed) in the third dimension. CUMULVS supports this mixing of decompositions. The global lower and upper bounds are used to specify the complete extent of the array (as if it were available in a single processor's memory). For arrays where a decomposition is across processors (*stvBlock, stvCyclic, stvExplicit*), the number of processors in that dimension needs to be given to CUMULVS. The first entry in pshape corresponds to the first axis that needs the information, and not necessarily to the first axis.

There are four decomposition types supported by CUMULVS.

stvBlock The array is block decomposed along this axis. The user may specify the block size. If the size of the array in this coordinate direction is not evenly divisible by the number of processors, then the block size is increased by one in the first j processors where $j = mod(size, nproc)$.

stvCyclic The array is cyclically decomposed along this axis. The user may specify the block size of the cycle to get the usual HPF-style block-cyclic decomposition. The default cycle size is 1.

stvCollapse The array is not decomposed along this axis. All indices reside on a single processor

stvExplicit The array is explicitly decomposed into non-uniform blocks in this coordinate direction. `axisInfo` contains the lower index (in global coordinates) that resides on this processor. `axisInfo2` contains the upper index (in global coordinates) that resides on this processor.

Examples

1. A 100×100 two-dimensional array is decomposed in a block manner in its *first* dimension on a linear array of 10 processors. The array indices are $\{1:10, 1:100\}$ on processor 0, $\{11:20, 1:100\}$ on processor 1, and so on.

```
dim = 2
axisType = {stvBlock, stvCollapse}
axisInfo = {stvDefault, stvDefault}
axisInfo2 = NULL
glb = { 1, 1 }
gub = { 100, 100}
prank = 1
pshape = 10
```

2. A 100×100 two-dimensional array is decomposed in a block manner in its *second* dimension on a linear array of 10 processors.

```
Same as above except
axisType = {stvCollapse, stvBlock}
```

3. A $100 \times 200 \times 300$ dimensional array is decomposed in a block manner in the first dimension with 4 logical processors in that dimension, block-cyclic with a block size of 25 in the second dimension across 4 logical processors and collapsed in the third dimension. This means that the array is decomposed over 16 processors. Hence, the processor with virtual address [0,1] has array indices $\{1:25, \{26:50, 126:150\}, 1:300\}$ while the processor with virtual address [3,3] has array indices $\{76:100, \{76:100, 176:200\}, 1:300\}$.

```
dim = 3
axisType = {stvBlock, stvCyclic, stvCollapse}
axisInfo = {stvDefault, 25, stvDefault}
axisInfo2 = NULL
glb = { 1, 1, 1 }
gub = { 100, 200, 300}
prank = 2
pshape = {4, 4}
```

Errors

Error conditions can be returned by `stv_decompDefine`.

Name	Possible cause
stvStatusBadParam	Invalid argument
stvStatusNoMem	unable to malloc memory

See Also

`stv_fieldDefine`, `stv_paramDefine`, `stv_sendToFE`

stvfdumplocalfieldbounds() stv_dumpLocalFieldBounds()

Prints out in a human readable form the global coordinate indices that are mapped to the calling processor of a field that has been described to CUMULVS using `stv_fieldDefine`.

Synopsis

C	`int info = stv_dumpLocalFieldBounds( int fieldId )`
FORTRAN	`stvfdumplocalfieldbounds( fieldId, info )`

Parameters

`fieldId`	–	integer specifying the Id of the decomposed field
`info`	–	integer error return value

Discussion

The routine `stv_dumpLocalFieldBounds` prints out the global indices that CUMULVS believes to reside in the calling process for a particular decomposed field. This is a useful debugging routine so that that programmer can verify agreement between the user's code and CUMULVS internals.

Examples

```
C:
   info = stv_dumpLocalFieldBounds( fieldId );

FORTRAN:
   call stvfdumplocalfieldbounds( fieldId, info )

Example Output:

   Global Array Bounds for Field "density":
       [ 1 - 60 ][ 1 - 60 ][ 1 - 120 ]

   Logical array of processors is:
       2  X  2

   Local processor address is:
       { 0, 0 }

   Local Portion of Global Space for Field "density":

       Bounds for Axis #1:
```

```
        [ 1 - 30 ]

    Bounds for Axis #2:
        [ 31 - 60 ]

    Bounds for Axis #3:
        [ 1 - 120 ]
```

Errors

Error conditions that can be returned by `stv_decompDefine`.

Name	Possible cause
stvStatusBadParam	Invalid argument, usually a bad field identifier

See Also

`stv_fieldDefine`, `stv_decompDefine`

stvffielddefine() stv_fieldDefine()

Defines a field of data to CUMULVS with a particular decomposition as set in the routine `stv_decompDefine`.

Synopsis

C

```
#include "stv.h"
int fieldId = stv_fieldDefine( void *var, char *name,
                               int decompId, int *arrayoffset,
                               int *arraydecl, int type,
                               int *paddr, int aflag )
```

FORTRAN

```
include 'fstv.h'
stvffielddefine(  var, name, decompId, arrayoffset,
                               arraydecl, type, paddr, aflag )
```

Parameters

`var`	–	pointer to a processors local storage for the decomposed array of data
`name`	–	string name of the variable so that a viewer program can identify the program variable as something meaningful.
`decompId`	–	integer identifier returned from the routine `stv_decompDefine`. Tells CUMULVS how this array has been decomposed across processors. Several arrays can share the same decomposition.
`arrayoffset`	–	integer array that indicates the offset, if any, to the first valid data element in the local storage array; used with arraydecl below to index into memory to select data elements. This array must be *dim* long as specified in the call to `stv_decompDefine`.
`arraydecl`	–	integer array that informs CUMULVS of the declared size of the local storage array so that it can properly index into memory to select data elements. This array must be *dim* long as specified in the call to `stv_decompDefine`.
`type`	–	integer specifying the data type of the field. The data type can be any of the CUMULVS standard data types (see below).
`paddr`	–	integer array that holds the address of this processor in the virtual processor array. This array should have *prank* entries as specified in the call to `stv_decompDefine`. Process address element i must be in the range $0..pshape_i$ where $pshape_i$ is element i of the array *pshape* as passed to `stv_decompDefine`.
`aflag`	–	integer option flag that is the sum of the desired options: *stvvisonly*
`fieldId`	–	returned integer handle that identifies this field.

Discussion

The routine `stv_fieldDefine` defines a field to CUMULVS. Fields are assigned string names so that they are easily identified on the viewer front-end. The data decomposition of the field must first be defined using the routine `stv_decompDefine`. CUMULVS must be told how the array storage has been declared by the program so it can properly extract data elements from the user's data space. CUMULVS must also be given the virtual processor address so that the system can properly match global storage coordinates to local storage coordinates.

The following data types are supported by CUMULVS:

CUMULVS type	C	FORTRAN
stvInt	int	integer*4
stvFloat	float	real*4
stvCplx	—	complex*8
stvDouble	double	real*8
stvDcplx	—	complex*16
stvLong	long	—
stvShort	short	integer*4
stvStr	char *	character
stvUshort	unsigned short	—
stvUint	unsigned int	—
stvUlong	unsigned long	—
stvByte	char	character

The local array storage offsets in arrayoffset are intended for use with any decompositions that are utilized with overlapping boundaries or other storage restrictions. For the case where there is no offset to the first valid data element in the storage area, there are predefined constants provided with CUMULVS to signify this default array offset:

C	FORTRAN
stvNoFieldOffsets	STVNOFIELDOFFSETS

Examples

1. The array p is 100×100 two-dimensional array that is decomposed in a block manner in its *first* dimension on a linear array of 10 processors. The following fragments will properly define this array to CUMULVS.

```
C:

float p[10][100];
int arrayDecl[2] = {10,100};
    int addr;

decompId = stv_decompDefine(dim, axisType, axisInfo, glb, gub,
                     prank, pshape);

    /* on processor n */
addr = n;
    fieldId = stv_fieldDefine( p, "pressure", decompId, stvFloat,
                     stvNoFieldOffsets, arrayDecl, &addr, stvVisOnly);

FORTRAN:
```

```
real p(10,100)
integer arrayDecl(2), addr
data arrayDecl /10,100/

call stvfdecompdefine(dim, axisType, axisInfo, glb, gub, prank,
                     pshape, decompId)
C     on processor n
    call stvffielddefine( p, "pressure", decompId, STVFLOAT,
                     STVNOFIELDOFFSETS, arrayDecl, n, STVVISONLY, fieldId)
```

Errors

Error conditions can be returned by `stv_fieldDefine`.

Name	Possible cause
stvStatusBadParam	Invalid argument. Usually an invalid decomposition Id.
stvStatusNoMem	unable to malloc memory

See Also

`stv_particleFieldDefine`, `stv_paramDefine`, `stv_sendToFE`

stvfparamdefine() stv_paramDefine()

Defines a scalar steering parameter for a CUMULVS front-end viewer to modify

Synopsis

C	`int paramId = stv_paramDefine( char *name, void * var, int type, int flag )`
FORTRAN	`stvfparamdefine( name, var, type, flag, paramId )`

Parameters

`name`	–	name of the parameter as it should appear to a front-end viewer
`var`	–	pointer to the variable where CUMULVS will deposit new parameter values.
`type`	–	identifies the type of parameter. See the `pvm_fieldDefine` for the mapping of CUMULVS types to C or Fortran types.
`flag`	–	identifies sub-actions for CUMULVS. Currently, only the flag stvVisOnly is recognized.
`paramId`	–	parameter Id that is used as a handle for the internal data structure that holds the parameter information.

Discussion

The routine `stv_paramDefine` allows a programmer to define a steering parameter. After each call to `stv_sendToFE`, a defined steering parameter may have been updated by a front-end viewer. CUMULVS insures that a parameter is updated at the same iteration (logical time) for each node in the parallel program. The routine `stv_sendToFE` returns the number of parameters that were changed. A programmer may call `stv_isParamChanged` to determine if a particular parameter has been updated.

Examples

```
C:
   paramId = stv_paramDefine( "Velocity", v, stvFloat, stvVisOnly );

FORTRAN:
   call stvfparamdefine( 'Velocity', v, STVFLOAT, STVVISONLY, paramid )
```

Errors

Error conditions that can be returned by `stv_paramDefine`.

Name	Possible cause
stvStatusNoMem	could not allocate memory for internal data structures

See Also

`stv_fieldDefine`, `stv_vParamDefine`, `stv_isParamChanged`, `stv_sendToFE`

stv_vparamDefine()

Defines a vector of steering parameter for a CUMULVS front-end viewer to modify. Available only from C.

Synopsis

C
```
int paramId = stv_vparamDefine( char *name, void ** vars,
                                char **pnames, int *types,
                                 int num, int aflag )
```

Parameters

`name`	–	name of the parameter as it should appear to a front-end viewer
`vars`	–	array of pointers to variables where CUMULVS will deposit new parameter values.
`type`	–	identifies the type of parameter. See the `pvm_fieldDefine` for the mapping of CUMULVS types to C or Fortran types.
`pnames`	–	array of parameter names as they should appear to a front-end viewer
`types`	–	array of types for parameters
`num`	–	number of parameters represented in this "vectored" parameter
`flag`	–	identifies sub-actions for CUMULVS. Currently, only the flag stvVisOnly is recognized.
`paramId`	–	parameter Id that is used as a handle for the internal data structure that holds the parameter information.

Discussion

The routine `stv_vparamDefine` allows a programmer to define a vector of steering parameters. The parameters in the vector are individually steerable but are referred to collectively by the name of the entire vector.

After each call to `stv_sendToFE`, a defined steering parameter may have been updated by a front-end viewer. CUMULVS insures that a parameter is updated at the same iteration (logical time) for each node in the parallel program. The routine `stv_sendToFE` returns the number of parameters that were changed. A programmer may call `stv_isParamChanged` to determine if a particular parameter has been updated.

Examples

C:

```
paramId = stv_vparamDefine( "State Vector", vars, names, types, num,
stvVisOnly );
```

Errors

Error conditions that can be returned by `stv_vparamDefine`.

Name	Possible cause
stvStatusNoMem	could not allocate memory for internal data structures

See Also

`stv_fieldDefine`, `stv_paramDefine`, `stv_isParamChanged`, `stv_sendToFE`

stvfisparamchanged() stv_isParamChanged()

Tells whether a parameter has been changed by a CUMULVS front-end viewer.

Synopsis

C	`int info = stv_isParamChanged( int paramId )`
FORTRAN	`stvfisparamchanged( paramId, info )`

Parameters

`paramId`	–	parameter Id handle as returned from `stv_paramDefine`.
`info`	–	0 if parameter is unchanged. Non-zero if parameter changed.

Discussion

The routine `stv_isParamChanged` tells the program if a particular parameter was changed during the most recent call to `stv_sendToFE`. Calling `stv_isParamChanged` resets the status variable. This means that a second call to this subroutine with the same parameter Id will always return false, unless `stv_sendToFE` has been called and the parameter was again updated.

Examples

```
C:
   changed = stv_isParamChanged( paramId );

FORTRAN:
   call stvfisparamchanged( paramid, changed )
```

See Also

`stv_isDefineParam`, `stv_sendToFE`

stvfparticledefine() stv_particleDefine()

Defines a "particle" composition domain for particle-based computations

Synopsis

C

```
int particleId = stv_particleDefine( char *name, int dim,
                                     int *glb, int *gub, int nproc,
                                     STV_GET_PARTICLE get_particle,
                                     STV_MAKE_PARTICLE make_particle )
```

FORTRAN

```
stvfparticledefine( name, dim, glb, gub, nproc,
                    get_particle, make_particle,
                    particleId )
```

Parameters

`name`	–	name of the particle domain as it should appear to a front-end viewer.
`dim`	–	integer specifying the dimension of the particle computational domain.
`glb`	–	integer array that specifies the global lower bounds of the particle computational domain.
`gub`	–	integer array that specifies the global upper bounds of the particle computational domain.
`nproc`	–	integer specifying the number of processors computing in the particle domain.
`get_particle()`	–	user provided routine for looking up particular particles in the computational domain (see below).
`make_particle()`	–	user provided routine for creating new particles in the computational domain, required for checkpointing only (see below).
`particleId`	–	returned particle Id that is used as a handle for the internal CUMULVS data structure that holds the particle domain information.

Discussion

The routine `stv_particleDefine()` allows a programmer to define a particle computation space for CUMULVS. This "particle" domain serves the same function as a decomposition does for a contiguous computational domain (see `stv_decompDefine()`), by describing the space within which particles can exist, and how CUMULVS can obtain information regarding specific particles. The "particle Id" returned by `stv_particleDefine()` is used in subsequent calls to `stv_particleFieldDefine()` to declare the actual data fields that exist within each particle.

Unfortunately, there are no standard, well-defined particle computational domain structures, and each user application can maintain collections of particle data in a unique manner. Therefore CUMULVS cannot provide any automated particle accessor functions like those used with contiguous decompositions. It is the responsibility of the user to provide an accessor routine, `get_particle()`, that will return a user-defined particle handle for CUMULVS.

The format of the user's `get_particle()` routine is:

```
C:          void  get_particle( int index, STV_REGION R,
                         STV_PARTICLE_ID *id, int *coords )

Fortran:    fgetparticle( index, R, id, coords )
```

The `index` value is intended to select a specific particle from the user's collection on a given application task. The `STV_REGION` defines a portion of the particle space in which to search for particles, and the index refers to which of the particles in that region is desired (by order of occurrence). The `stv_particle_in_region()` routine assists the user by indicating whether the coordinates of a particle lie in the desired region. The search order of the particles in the region is arbitrary, and at the discretion of the user.

The `STV_PARTICLE_ID` returned by `get_particle()` should be some user-defined value that can be used subsequently to identify the given particle when accessing its data field values. The `STV_PARTICLE_ID` could be an integer index into an array of particles, or perhaps a pointer to a particle structure, at the user's discretion. The user must also set the values of the CUMULVS-provided integer array `coords` to hold the coordinates of the returned particle.

It should be noted that, for efficiency only, CUMULVS will initially invoke `get_particle()` with an index value of `-1` when beginning to collect each new set of particle Ids. This is intended as a convenience for the user, to allow the saving and resetting of search state in `get_particle()` routines. CUMULVS will always request particles with increasing index values, so better performance can be obtained by beginning each search where the previous search had stopped. CUMULVS ignores the values returned for a `-1` index value.

In addition to a `get_particle()` routine, the user can define an additional `make_particle()` routine, for future releases of CUMULVS which will support checkpointing. This (currently optional) routine will be used when CUMULVS needs to "load" the checkpoint data into user application tasks to restart them from a checkpoint. The format of the user's `make_particle()` routine is:

```
C:          void  make_particle( int *coords,
                       STV_PARTICLE_ID *id )

Fortran:    fmakeparticle( coords, id )
```

Here, CUMULVS passes in an array of coordinates and the user is expected to allocate or initialize a new particle which is located at those given coordinates. The user-defined handle for this new particle should be returned in the `STV_PARTICLE_ID id`.

Examples

```
C:
   void  get_helicopter __STV_Prototype__(( int, STV_REGION,
           STV_PARTICLE_ID *, int * ));

   particleId = stv_particleDefine( "Helicopter", 3, glb, gub, nprocs,
           get_helicopter, (STV_MAKE_PARTICLE) NULL );

   void get_helicopter( hnum, R, id, coords )
   int hnum;
   STV_REGION R;
   STV_PARTICLE_ID *id;
   int *coords;
   {
       static HELI H_save = (HELI) NULL;
       static int count_save = -1;

       HELI H;

       int hcoords[3];
       int count;

       /* Reset */

       if ( hnum == -1 )
       {
           *id = (STV_PARTICLE_ID) NULL;
```

```
    H_save = (HELI) NULL;
    count_save = -1;
}

/* Restore */

if ( hnum >= count_save && count_save != -1 )
{
    H = H_save;

    count = count_save;
}

/* Fresh Search */

else
{
    H = HELIS;

    count = 0;
}

/* Search for Helicopter */

while ( H != NULL )
{
    hcoords[0] = H->loc_x;
    hcoords[1] = H->loc_y;
    hcoords[2] = H->loc_z;

    if ( stv_particle_in_region( R, hcoords, 3 ) )
    {
        if ( ++count == hnum )
        {
            *id = (STV_PARTICLE_ID) H;

            coords[0] = hcoords[0];
            coords[1] = hcoords[1];
            coords[2] = hcoords[2];
```

```
                    H_save = H->next;
                    count_save = count;

                    return;
                }
            }

            H = H->next;
        }

        /* Helicopter Not Found */

        *id = (STV_PARTICLE_ID) NULL;

        H_save = (HELI) NULL;
        count_save = -1;
    }
```

```
FORTRAN:
   call stvfparticledefine( 'Helicopter', 3, glb, gub, nprocs,
       get_helicopter, 0, particleid )

   subroutine get_helicopter( hnum, R, id, coords )
   integer hnum;
   integer R(6);
   integer id;
   integer coords(3);
   . . .
```

Errors

Error conditions that can be returned by `stv_particleDefine()`.

Name	Possible cause
stvStatusNoMem	could not allocate memory for internal data structures
stvStatusBadParam	invalid values passed or required arguments omitted

See Also

`stv_particleFieldDefine()`, `stv_particle_in_region()`, `stv_decompDefine()`

stvfparticlefielddefine() stv_particleFieldDefine()

Defines a data field within a "particle" composition domain

Synopsis

C

```
int pfieldId = stv_particleFieldDefine( char *name,
                              int particleId, int type,
                              STV_GET_PFIELD get_pfield,
                              STV_SET_PFIELD set_pfield,
                              int aflag )
```

FORTRAN

```
stvfparticlefielddefine( name, particleId,
                              type, get_pfield, set_pfield,
                              aflag, pfieldId )
```

Parameters

`name`	–	name of the particle data field as it should appear to a front-end viewer.
`particleId`	–	particle computational domain handle as returned from a call to `stv_particleDefine()` or `stvfparticledefine()`. Provides bounds of the computational domain for CUMULVS. Each particle domain can have any number of data fields defined for it.
`type`	–	integer specifying the data type of the particle data field. The type can be set to any of the CUMULVS standard data types (see below).
`get_pfield()`	–	user provided routine for extracting the data value for this data field within a particle (see below). The specific particle is identified by a user-defined handle as returned by an invocation of the user's `get_particle()` routine (see `stv_particleDefine()`.
`set_pfield()`	–	user provided routine for setting the value of this data field in a particle, required for checkpointing only (see below).
`aflag`	–	integer option flag that is the sum of the desired options. Currently the only option available is `stvVisOnly` / `STVVISONLY`, but CUMULVS checkpointing will ulitmately utilize the `stvCpOnly` / `STVCPONLY` options as well.
`pfieldId`	–	returned particle field Id that identifies this field.

Discussion

The routine `stv_particleFieldDefine()` allows a programmer to define individual data fields for the particles in a particle computation space for CUMULVS. Each particle will have values for each defined particle data field. CUMULVS allows access to any desired subset of particle fields for visualization. The "pfield Id" returned by `stv_particleFieldDefine()` identifies the data field for any other operations or options.

The data field can be of any CUMULVS data type. The following data types are currently supported:

CUMULVS type	C	FORTRAN
stvInt	int	integer*4
stvFloat	float	real*4
stvCplx	—	complex*8
stvDouble	double	real*8
stvDcplx	—	complex*16
stvLong	long	—
stvShort	short	integer*4
stvStr	char *	character
stvUshort	unsigned short	—
stvUint	unsigned int	—
stvUlong	unsigned long	—
stvByte	char	character

Unfortunately, there are no standard, well-defined particle computational domain structures, and each user application can maintain collections of particle data in a unique manner. Therefore CUMULVS cannot provide any automated particle data accessor functions like those used with contiguously decomposed data fields. It is the responsibility of the user to provide an accessor routine, `get_pfield()`, that will return the actual value of a data field for CUMULVS.

The format of the user's `get_pfield()` routine is:

```
C:        void  get_pfield( STV_PARTICLE_ID id, STV_VALUE *value )

    Fortran:  fgetpfield( id, value )
```

The `STV_PARTICLE_ID id` is a user-defined particle handle returned by a call to the user's `get_particle()` routine for the given particle domain (see `stv_particleDefine()`). CUMULVS allocates the proper amount of memory for the generic `STV_VALUE value` pointer, so the user's `get_pfield()` routine need only assign it the proper value before returning.

In addition to a `get_pfield()` routine, the user can define an additional `set_pfield()` routine, for future releases of CUMULVS which will support checkpointing. This (currently optional) routine will be used when CUMULVS needs to "load" the checkpoint data into user application tasks to restart them from a checkpoint. The format of the user's `set_pfield()` routine is:

```
C:        void  set_pfield( STV_PARTICLE_ID id, STV_VALUE *value )

    Fortran:  fsetpfield( id, value )
```

Here, CUMULVS passes in the generic `STV_VALUE value` which holds the checkpoint data for the given particle data field. The user is expected to assign that value to the data field for the particle referenced by `STV_PARTICLE_ID id`.

Examples

```
C:
   void  get_helicopter_heading __STV_Prototype__(( STV_PARTICLE_ID,
           STV_VALUE * ));

   particleId = stv_particleDefine( "Helicopter", ... );

   pfieldId = stv_particleFieldDefine( "Heading", particleId,
           stvDouble, get_helicopter_heading, (STV_SET_PFIELD) NULL,
           stvVisOnly );

   void get_helicopter_heading( id, value )
   STV_PARTICLE_ID id;
   STV_VALUE value;
   {
       *value = (STV_VALUE) &( ((HELI) id)->heading );
   }


FORTRAN:
   call stvfparticledefine( 'Helicopter', ..., particleId )

   call stvfparticlefielddefine( 'Heading', particleId, STVDOUBLE,
       get_helicopter_heading, 0, STVVISONLY )
```

```
subroutine get_helicopter_heading( id, value )
integer id;
real*8 value

value = heli_heading( id )

return

end
```

Errors

Error conditions that can be returned by `stv_particleFieldDefine()`.

Name	Possible cause
stvStatusNoMem	could not allocate memory for internal data structures
stvStatusBadParam	invalid values passed or required arguments omitted

See Also

`stv_particleDefine()`, `stv_fieldDefine()`

stvfsendtofe() stv_sendToFE()

Listen for new viewing front-ends to attach to a program. Send data to any attached front ends. Get steering parameters from front ends.

Synopsis

C	`int info = stv_sendToFE( )`
FORTRAN	`stvfsendtofe( info )`

Parameters

`info` – 0 if parameters have not changed. Positive return value indicates that parameters have changed. Negative value is an error.

Discussion

The routine `stv_sendToFE` handles all of the details of attaching a front-end viewer to a program, sending data to a front-end and getting steering parameters back from a front-end viewer. Fields must have been defined by `stv_fieldDefine` and parameters must have been defined by `stv_paramDefine`. This routine returns the number of user-defined steering parameters that have been updated during the call. Negative values indicate an error.

This routine must be called by all nodes in a parallel program, whether or not data is being sent. CUMULVS uses a loosely synchronous model of computation with flow control. This means that `stv_sendToFE` will not block a task unless data from a previous iteration has not been transmitted completely to an attached front-end by all node programs. Otherwise, CUMULVS does not explicitly synchronize tasks in `stv_sendToFE`.

If a front-end viewer detaches at any time from a running computation, a task will not block in `stv_sendToFE`.

Examples

```
C:
   nchanged= stv_sendToFE();

FORTRAN:
   call stvfsendtofe( nchanged )
```

See Also

`stv_paramDefine`, `stv_fieldDefine`, `stv_isParamChanged`

stvfversion() stv_version()

Return the version of CUMULVS that is being used.

Synopsis

C	`char * verstring = stv_version()`
FORTRAN	`stvfversion( verstring )`

Parameters

`verstring` – string that contains the current CUMULVS version number.

Discussion

The routine `stv_version` returns a string containing the version number of CUMULVS. The most recent version number is at least "1.0.0".

Examples

```
C:
   verstring = stv_version();

FORTRAN:
   call stvfversion( verstring )
```

7. Viewer Library Man Pages

Man Pages to be added:

stv_viewer_init()

stv_clear_view_field_select()

stv_get_view_field_name()

stv_viewer_request_field()

stv_allocate_Field_arrayDecl()

stv_allocate_Field_Data()

stv_free_field_array()

stv_reset_Field_Particles()

stv_viewer_receive_frame()

stv_viewer_send_XON()

stv_viewer_set_VisRegion()

stv_viewer_set_VisFrequency()

stv_dump_field_array()

stv_dump_vfg_field_array()

stv_dump_user_field_array()

stv_viewer_send_FieldHaltGroup()

stv_viewer_send_FieldHaltAll()

7.1. Viewer Steering Library Man Pages

Man Pages to be added:

stv_viewer_steering_init()

stv_get_view_param_name()

stv_viewer_steering_request()

stv_viewer_steer_parameter()

stv_viewer_steer_vparameter()

stv_viewer_send_NewParams()

stv_viewer_steering_release()

7.2. Viewer Utility Library Man Pages

Man Pages to be added:

stv_copy_converted_data()

stv_str_of_typed_data()

stv_init_region()

stv_copy_region()

stv_init_cell()

stv_copy_cell()

stv_memcheck()

stv_copy_str()

stv_setopt()

stv_getopt()

7.3. Message Passing Library Man Pages

Man Pages to be added:

stv_myid()

stv_mp_err()

Somebuddy gots ta do dis.

. . .