CONF-9608140--1

CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications *

G. A. Geist, II James Arthur Kohl Philip M. Papadopoulos †

Abstract

The use of visualization and computational steering can often assist scientists in analyzing large-scale scientific applications. Fault-tolerance to failures is of great importance when running on a distributed system. However, the details of implementing these features are complex and tedious, leaving many scientists with inadequate development tools. CUMULVS is a library that enables programmers to easily incorporate interactive visualization and computational steering into existing parallel programs. The library is divided into two pieces: one for the application program and one for the, possibly commercial, visualization and steering front-end. Together these two libraries encompass all the connection and data protocols needed to dynamically attach multiple independent viewer front-ends to a running parallel application. Viewer programs can also steer one or more user-defined parameters to "close the loop" for computational experiments and analyses. CUMULVS allows the programmer to specify user-directed checkpoints for saving important program state in case of failures, and also provides a mechanism to migrate tasks across heterogeneous machine architectures to achieve improved performance. Details of the CUMULVS design goals and compromises as well as future directions are given.

1 Introduction

Scientists developing large-scale distributed scientific applications face many unique problems. Such applications need to be monitored at several different levels. During the debugging stage, for example, a programmer may want to view a program's use of message passing primitives and visually monitor how distributed data is being modified. Once the application runs smoothly, the scientist may wish to examine the progress of the overall computation to insure that the results are being generated as expected. Using visualization to explore the computational domain can provide an intuitive analysis, especially for physically-based simulations. Being able to visualize intermediate values in the computational domain, while the application is still running, can be extremely useful for revealing algorithm dynamics and identifying subtle errors.

*Research supported by the Applied Mathematical Sciences Research Program of the Office of Energy Research, U.S. Department of Energy, under contract DE-AC05-96OR22464 with Lockheed Martin Energy Research Corporation

†All authors are with the Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN, 37831-6367

Beyond simply observing a running application, the scientist may also wish to interactively control it. The scientist might explore a "what if" analysis where parameters of the computation are adjusted, perhaps on-the-fly, to gain understanding of some underlying principle. This is known as "computational steering." Computational steering has the potential to revolutionize computer simulation experiments by allowing scientists to interactively explore (steer) a simulation in time and/or space, and concentrate more on the science than on the computer. Through the use of such interaction the computer will become a more useful tool to the engineer, allowing real time exploration of a design space. Interactive steering supersedes the traditional simulation mode of many long-running experiments, which may not produce the desired results. Instead computational steering allows the scientist or engineer to "close the loop" and respond to simulation results as they occur by interactively manipulating the input parameters.

It is also critical to apply some method of failure recovery when executing long-running applications in a distributed system. Faults can occur in a variety of ways, including machine crashes and network overloading or failure. The user application may need to be reconstructed, or in the worst case completely restarted. To avoid catastrophic losses, the application needs to be able to roll-back to some previously saved state, or "checkpoint," and continue on from that point rather than starting from scratch.

Unfortunately, the efficient handling of the above issues requires a special expertise in computer science and a level of effort higher than the typical application scientist is willing to expend. CUMULVS provides a robust mechanism for interactively visualizing and steering a running application, and allows user-directed checkpointing and migration of application tasks. CUMULVS is a library middle-ware that bridges the gap between existing application codes and commercial visualization packages, allowing programmers to add real-time visualization and interactive steering to their parallel simulations. The interactions between the user application and the "front-end" viewing and steering interfaces are dynamic and fault-tolerant, and can be initiated or terminated on-the-fly. There can be any number of simultaneously attached front-end "viewers[1]."

Using CUMULVS, the programmer simply declares how an array or field of variables has been decomposed on a collection of parallel processors, and specifies which parameters are allowed to be modified or "steered" during the computation. Then, at the point in the iterative calculation where these values are valid (as indicated by the placement of a single CUMULVS call) these variables can be reliably read and updated based on instructions from the front-end viewer.

Much of the same infrastructure that is used for interactive visualization and steering can be used for *user-directed* checkpointing. The descriptions of the data and its decompositions can be utilized to efficiently collect the checkpoint information, and to restore the application in the event of a failure. Further, these checkpoints can be used to improve performance by interactively migrating application tasks, even across heterogeneous machine architectures. It should be noted that heterogeneous migration is not possible with automatic system-directed checkpointing, where full core images are saved. Also, because the user decides precisely what data CUMULVS needs in its checkpoints, the amount of data collected can be significantly smaller.

---

[1] "Viewer" is a generic phrase to describe a program for visualizing *or* steering an application.

While the checkpointing in CUMULVS is still in the "experimental" stages of development, the integration of steering and visualization with commercial (such as AVS) and user-programmed graphical front-ends is well developed with support for a wide variety of parallel data decompositions. The system has been designed to provide a high degree of flexibility while remaining efficient.

CUMULVS was designed from the outset to be dynamic wherever possible. For example, multiple viewers can start-up, interactively "attach," and independently view different fields or regions of the same parallel computation. These same independent viewers can also coordinate updates to steering parameters. To prevent incoherent parameter updates, CUMULVS requires viewers to acquire a token before modifying an application parameter. CUMULVS handles all these seemingly mundane details efficiently and robustly, even in the face of viewing front-ends that can attach and detach from the parallel application at any time.

Because network bandwidth is always a scarce resource, front-end viewers can specify a region for visualization, including the granularity of the desired data. To reduce network traffic CUMULVS downsizes these datasets in the parallel application itself, instead of at the viewer. CUMULVS-capable viewers are sent only the data that they have requested. CUMULVS is also very low overhead when no viewer is currently connected: only a single message probe is issued once per iteration to check for new viewer or steering connections.

The remainder of this paper is divided into four sections. Section 2 overviews the user interface and some of the connection protocols used in CUMULVS. Section 3 details the interface for creating custom CUMULVS-capable viewers (a standard AVS viewer already exists and is distributed with the software). Section 4 overviews how fault-tolerance and checkpointing issues are being addressed by CUMULVS. Section 5 addresses some of the issues for the future of the project.

2 CUMULVS Overview and User Interface

The CUMULVS library provides several important features for the computational scientist. It handles all of the details of collecting and transferring distributed data fields to the viewers and oversees adjustments to steering parameters in the application. The complete system also manages all aspects of the dynamic attachment and detachment of viewers to a running simulation.

CUMULVS applications need not always be connected to a given viewer, and multiple viewers can be attached / detached interactively as needed. This proves especially useful for long-running applications that may not require constant monitoring. Though CUMULVS's primary purpose is manipulating and collecting data from distributed or parallel applications, it is also useful with serial applications for the purpose of transferring data from the computation engine over a network to a visualization front-end.

When a CUMULVS viewer attaches to a running application, it does so in terms of a "data field request." This request includes some set of data fields, a specific region of the computational domain to be collected, and the frequency of data "frames" which are to be sent to the viewer. CUMULVS automatically handles the collection of a sub-region, or visualization region, of data from the application. The viewer requests this region from

the application via a request message that is handled by the user library routines. For a distributed or parallel application, each task in the distributed simulation need only identify its logical processor position in a data field decomposition, then CUMULVS can determine which data elements are present in each task. Several data decompositions are supported, including generic block-cyclic decompositions as described by HPF [9, 1], and particle-type decompositions with user-defined accessor functions.

In addition to the boundaries of the sub-region, a full visualization region specification also includes a "cell size" for each axis of the computational domain. The cell size determines the stride of elements to be collected for that axis, e.g. a cell size of 2 will obtain every other data element. This feature provides for more efficient high-level overviews of larger regions by using only a sampling of the data points, while still allowing every data point to be collected in smaller regions where the details are desired.

Once CUMULVS has collected each local task's data for a given visualization region, the data is sent to the viewer task where it is automatically assembled into a coherent frame for animation. CUMULVS ensures that each viewer has a time-coherent view of the parallel data. An internal sequence number is maintained to determine where each application task is in the computation, and any attached viewers loosely synchronize with the application for each data frame they receive. The loose synchronization means that a viewer can determine on which iteration tasks are computing (within some desired number of iterations, as controlled by the frame frequency) without using an explicit barrier. The frequency of data frames can be set from the viewer, so the user can adjust how often frames are sent from the application, thereby reducing overhead and the loose synchronization effects.

CUMULVS supports coordinated computational steering of applications by multiple collaborators. A token scheme prevents conflicting adjustments to the same steering parameter by different users. Consistency protocols are used to verify that all tasks in a distributed application apply the steering changes in unison. So, scientists, even if geographically separated, can work together to direct the progress of a computation without concern for the consistency of steering parameters among distributed tasks.

CUMULVS can be utilized on top of any complete message-passing communication system, and with any front-end visualization system. Current applications use PVM as a message-passing substrate, and several visualization systems are supported including AVS and TCL/TK. Porting CUMULVS to a new system requires only creation of a single declaration file, to define the proper calling sequences for CUMULVS.

The current CUMULVS system evolved from an earlier prototype system that linked a PVM application to AVS for floating point data visualization and some simple steering operations [7]. CUMULVS completely generalizes this early system and now supports all standard primitive data types with built-in type conversion when desired. CUMULVS also provides a fault-tolerant communication protocol so that failures in either an application or a viewer can be gracefully handled.

While on the surface the concept of collecting data from an application, or of passing steering parameters to an application, may seem rather straightforward, there are many underlying issues that make such a system difficult to construct. Creating CUMULVS in its current form required the development of a variety of synchronization protocols to maintain consistency among the many distributed application tasks without introducing any deadlock

conditions. These protocols also had to be dynamic to allow viewers to attach at will, and yet had to be tolerant of faults and failures. Efficient, general algorithms had to be formulated for the packing and unpacking of data in different data decompositions — obtaining every "Nth" element within a sub-region becomes significantly more complicated when working with arbitrarily mixed block and cyclic decompositions. Finally, the viewer/application interfaces had to be generalized to support a variety of viewers with different data and synchronization requirements. The end result is a system that automatically and efficiently handles all of these challenging details with a minimal amount of user specification or effort.

2.1 User Library Interface

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 CUMULVS library consists of approximately 20,000 lines of C code, and can be integrated into applications written in either C or Fortran. Existing programs require only slight modifications to describe how particular data fields have been decomposed and which parameters can be steered by a viewer.

The following pseudo-code illustrates the typical statement sequence that a programmer would follow to define distributed data fields, steerable parameters, and enable visualization.

1. Initialize CUMULVS data structures (`stv_init()`)
2. Define data decomposition (`stv_decompDefine()`)
3. Define data field with a previously defined decomposition (`stv_fieldDefine()`)
4. Define steering parameters (`stv_paramDefine()`)
5. Start main iterative loop
   - <usual calculation>
   - *nchanged* = `stv_sendToFE()`
   - <program response to *nchanged* steered parameters>
6. End of main iterative loop

Figure 1: Typical execution order for a CUMULVS program

The predominant complication is getting 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 sub-

routine 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()` in the same place in each parallel task.

CUMULVS understands a variety of standard decomposition types, including 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)
- The global upper and lower bounds of the data array
- The dimension of logical processor decomposition
- How each axis of the array is decomposed

The data is assumed to be 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.

2.2 Example

To get a flavor for the changes required to an existing program, Figure 2.2 shows the actual calls needed to define and send a pressure field in a parallel acoustic wave propagation application. The order is very simple: initialize the CUMULVS internals; define a data decomposition (handle returned in decompId); define a field with the described data decomposition (handle returned in fieldId); and, at the end of the computational loop, send fields to attached viewers.

The additional call at the end of the logical computational loop directs when data can be sent to a viewing front-end. In the case where no viewers are attached to the running simulation, this single call results in a single message probe to check for requested connections.

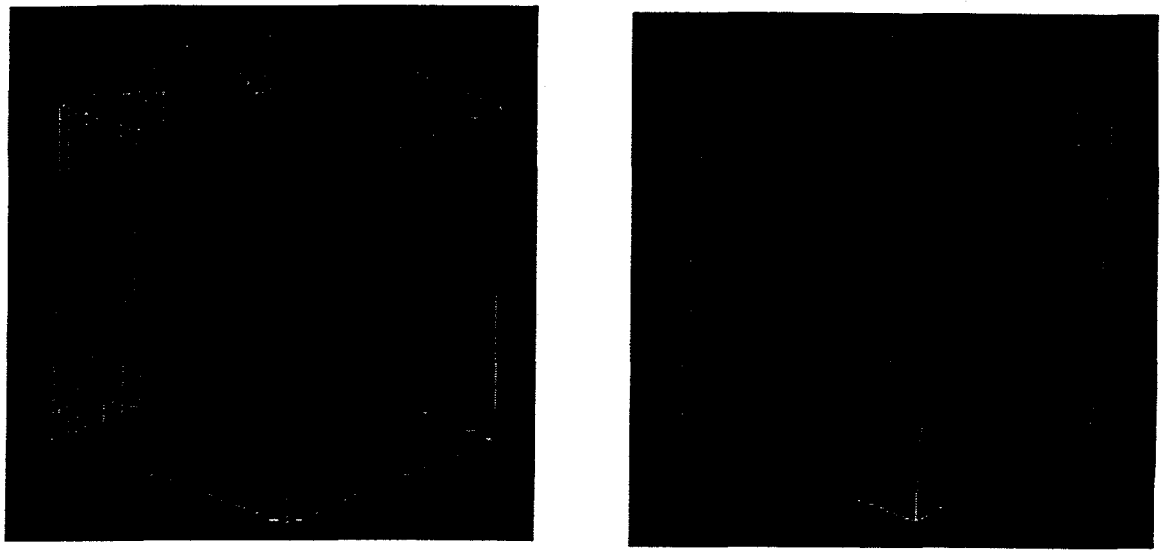
Using CUMULVS, the programmer of this simulation is able to adjust input parameters, such as seismic shot location and intensity, on-the-fly. Figure 2.2 shows frames as they would appear from two different geographically separated viewers visualizing the same simulation. From "Site A" the view covers the entire computational region at a low level of detail to provide a coarse overview of the progress of the simulation. The "Site B" view shows a detailed close-up of a particular area of the shock wave reflection, showing every data point in that region.

3 Viewer and Steering Interface

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

```
C - Initialize CUMULVS Structures
      call stvfinit( SEISMICGRP, STVTAG, nproc, inst, info)

C - Define the Decomposition type
      call stvfdecompdefine(3,axisType,axisInfo,axisInfo1,glb,gub,
     +                 prank,pshape,decompId)

C - Define Pressure field with above decomposition
      call stvffielddefine(u1,'pressure',decompId,
     +    declare, STVFLOAT, paddr, STVVISCP, fieldId)

C - Top of main Computational Loop
      do 100 npdt = tmp_npdt,ntps,2

   . . .

C - Send visualization information to any attached front-ends
        call stvfsendtofe(info)

100   continue
```

Figure 2: CUMULVS Library Calls in a Parallel Acoustic Wave Simulation



Site A – Coarse Overview, Entire Region

Site B – Detail of Shock Reflection

Figure 3: Parallel Acoustic Wave Propagation Simulation Viewers

simulation. These viewers should be sufficient for typical user needs, and special custom viewers need not be created. However, if a special viewer is desired, CUMULVS provides a simple interface to construct a custom user viewer.

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 viewer library 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, including the collection of data field values computational steering. The following subsections describe the viewer library in more detail.

3.1 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()`[2]. This function initializes a link with a specific user application and gathers information about the data fields and parameters that are

[2]Note that all CUMULVS library routines have an "STV" prefix, for historical reasons.

available from that application. This call 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 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 data field elements can be chosen for a data field request. The data fields are obtained by name and each can then be selected by setting its `selected` flag. The set of fields selected for a particular field request is considered a "view field group."

The viewer can select a specific data type for each data field in a view field group (VFG). The type for a given data field can be set to any of the supported CUMULVS data types, even if 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 (`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." The visualization 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 visualization 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 visualization 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 visualization region specifies the collection area for the entire group of data fields in a VFG. If different visualization 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.

After the desired data fields have been selected and the visualization region has been specified, a field request is sent to the application tasks using a single call to the

`stv_viewer_request_field()` routine. This routine returns a VFG instance that represents the group of data fields requested. The VFG instance 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.

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 visualization 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 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. 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 visualization 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 visualization region. CUMULVS takes care of verifying that the next data frame is collected using the proper visualization region, and will return a `stvStatusBadFrame` return code for the next frame 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.

3.2 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.

A viewer can initiate steering with a particular application by invoking 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.

Once steering has been successfully initialized, a specific steering parameter can be controlled by acquiring the appropriate steering token. Parameters are identified by name, as defined by the user application. The steering token for a particular parameter is obtained using the `stv_viewer_steering_request()` routine. If the token for a parameter is not already in a viewer's possession, then the `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 the `token` will be set instead to `stvSteerRqstd`. This 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.

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 a data value pointer, over into the viewer parameter structure, and then set 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 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.

4 Fault-tolerance Design

Parallel programming is already a complex task. Yet, current and future "parallel" machines will likely take on a more distributed character making this task more difficult. Parts of the user's computing engine may slow down, time-out, or simply fail. Instead of programming a single parallel machine, the user will have to distribute tasks across multiple independent machines and across multiple architectures. Message passing systems such as MPI ([8], [10]) and PVM ([5]), allow users to gather several different computers into a single virtual machine. While PVM in particular has hooks for creating fault-tolerant applications, a great deal of effort is required on the part of programmer to achieve fault-tolerance.

CUMULVS handles many, but not all, of the difficulties associated with creating fault-tolerant applications. For example, much of logic needed to reliably and correctly restart a failed parallel application has been moved to a separate process (one per machine) called a "checkpointing daemon" (cpd). The programmer must specify what variables need to be saved and provide logic to determine if the application is starting normally or from a checkpoint. CUMULVS manages the details of retrieving the most current (coherent) checkpoint and loading it into the user's variables. This so-called *user-directed* checkpointing requires more work by the programmer. However, there are two major benefits to this extra effort: checkpoints are generally smaller because only the essential data is saved; and, enough information is specified to allow a program to be migrated across architectures. Experimental versions of the checkpointing software have already demonstrated a "real-time" cross-platform migration of several parallel programs.

4.1 Design Issues

After extensive experimentation with steering and visualization using CUMULVS, it became evident that a large part of the application programmer's contribution was simply describing how data was stored in the parallel program. Often, the data that the user wanted to visualize or steer was the same data that needed to be saved in a checkpoint. Furthermore, the same descriptions could be used for both. By asking the programmer to describe the essential data needed for a program restart, the first step could be made in cross-platform migration and heterogeneous restarts of parallel programs. The primary design goal was to

make checkpointing and restarting the application a simple task for the programmer, while still allowing this cross-platform migration.

Many fault-tolerant application environments such as CoCheck [4], Isis [3], and Totem [2] are designed for single architecture programs. CoCheck works with PVM to save the entire binary image of a program and move it to another similar machine. The binary dump in CoCheck makes it impractical to migrate codes when a moderate number of compute nodes have failed. Isis and Totem use the concept of "virtual synchrony" to greatly simplify the logic of writing fault-tolerant programs, but requires either a partial or total ordering of all messages in the parallel program. This total ordering exacts a high overhead and is impractical for large numbers of nodes. In fact, Isis is particularly well suited for transaction systems where an event is either recorded by every node or none of the nodes. Their approach essentially gives a program the chance to roll-back to the last received message. CUMULVS, on the other hand, requires the program to roll-back to the last checkpoint. This makes CUMULVS well suited to large scientific applications where the desire is to limit the amount of lost computing cycles. However, it is unsuitable for *critical* applications such as bank transaction systems.

The design operates under the assumption that machines are, in fact, fairly stable and that a program should "pay" for fault-tolerance only when there is an actual failure. Checkpointing in any system is a relatively time consuming. In CUMULVS, the user directs when (how often) their program needs to save state to control how much overhead is incurred. When a code fails, all computation that occurred after the most recent checkpoint is lost. The entire application is rolled-back to the most recent checkpoint and then restarted. The user needs to structure the program logic so that their code can restart with the old data and *empty* message queues.

4.2 The Checkpointing Daemon

The current CUMULVS design has a separate checkpointing daemon (cpd) on each machine in the virtual machine. This collection of daemons makes up a dynamic fault tolerant program that is separate from any user's code. From an application's perspective, the cpd provides two basic functions:

1. Saving a checkpoint from an application
2. Loading a checkpoint into an application

In addition, the cpd:

1. Monitors the application for failures
2. Adds new computing resources in the event of machine failure
3. Signals non-failed nodes that the application should restart
4. Handles the migration of checkpoint data and tasks, if needed
5. Restarts complete parallel applications after a failure

There are two ways in which an application can respond to a failure, kill all nodes on any failure and perform a complete reload, or signal active nodes that they should load from a checkpoint. The first method requires the programmer to check at start up if data should be loaded from a checkpoint. The second method requires the programmer to check at every message for a restart. CUMULVS supports this second mode of operation and will flushes all old messages whenever a code restarts from a checkpoint. In either case, the cpd does the signaling and task management to properly restart a partially or completely failed parallel application.

4.3 Checkpoint Specifics

The predominant overhead in checkpointing is spent during the actual commitment of checkpoints. CUMULVS uses an asynchronous scheme where each task writes a checkpoint when the code makes a call to `stv_checkpoint()`. The application code does not explicitly synchronize at a checkpoint. However, a task will be blocked until the previous checkpoint has finished. It is the responsibility of the cpd's to make sure that a parallel task is restarted from a *coherent* checkpoint, that is, a checkpoint that corresponds to the same logical time step. Because programs are not explicitly synchronized, it is possible for the most recent checkpoint to be incomplete. If a failure occurs while in this state, then the cpd's must collectively revert to the last complete checkpoint.

An important issue is what level of data replication should be supported in the checkpoints. In the case of small checkpoints and a small number of machines, it is feasible to replicate the entire checkpoint data on each machine. This gives the highest degree of fault-tolerance because only one machine's data must be retrievable to restart a program. On the other hand, if the checkpoint data is very large, then data replication using standard low-speed networks is clearly impractical. Experiments are continuing with methods of specifying the required level of data redundancy.

The predominant overhead of checkpointing is the time taken to write data to disk or other non-volatile storage. If replication of checkpoint data is desired, then inter-machine bandwidth is also consumed to copy data from one machine to another. The cpd's also impose a small computational overhead. Currently, tasks pack and send checkpoint data to the local cpd and have it save the data on behalf of all tasks. This method will be replaced and each task will write its data to a file. The scheme will allow the use of parallel file I/O on systems that support it.

The basic checkpointing and restarting logic of the current system operates correctly. However it is not nearly as efficient as it could be and does not support enough options to be used in both large and small programs. A better monitoring and control system for the checkpoint data is is needed so that users can determine how much overhead is being consumed for fault-tolerance.

5 Future Directions

CUMULVS is an effective and straightforward system that allows scientists to interactively visualize and steer existing parallel computations. Furthermore, CUMULVS is flexible

enough to allow several geographically- separated scientists to collaborate by simultaneously viewing the same ongoing simulation. In addition, the checkpointing capability provided in CUMULVS simplifies the task of constructing reliable large-scale distributed applications.

There are several areas of future work to be explored with CUMULVS. Currently, an N-node parallel application can be retarted only with the same number of nodes. However, with the existing user data descriptions and checkpointing in CUMULVS, it will be possible to redistribute a parallel application across an arbitrary number of nodes.

The current viewer library provided in CUMULVS assumes that the viewer programs themselves are serial. There may be some benefit to coordinating parallel viewers for observing large parallel applications. This would require substantial protocol changes to produce an efficient, robust and user-friendly system.

In the short term, CUMULVS will be ported to a wider variety of visualization and interface systems. Alternate message-passing systems will also be explored. Currently, MPI-1 does not support the necessary functionality for the dynamics associated with CUMULVS. MPI-2, however, may provide a sufficient interface for CUMULVS.

References

[1] *High Performance Fortran Language Specification, Version 1.1,* Rice University, Houston, TX, November, 1994.

[2] D.A. Agarwal, "Totem: A Reliable Ordered Delivery Protocol for Interconnected Local Area Networks," PhD. Dissertation, Dept. of ECE, University of California, Santa Barbara, August 1994.

[3] K.P. Birman and R. Van Rennesse,, "Reliable Distributed Computing Using the Isis Toolkit", *IEEE Computer Society Press*, 1994.

[4] G. Stellner and J. Pruyne, "Providing Resource Management and Consistent Checkpointing for PVM", *1995 PVM User's Group Meeting* , Pittsburgh, PA.

[5] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, *PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, The MIT Press, 1994.

[6] A.S. Grimshaw, W.A. Wulf, J.C. French, A.C. Weaver, and P.F. Reynolds, Jr., "A Synopsis of the Legion Project," University of Virginia, Technical Report No. CS-94-20, June, 1994.

[7] J. A. Kohl, P. M. Papadopoulos, "A Library for Visualization and Steering of Distributed Simulations using PVM and AVS," Proc. of High Performance Computing Symposium. Montreal, Canada, pp. 243–254, 1995.

[8] Message Passing Interface Forum. Mpi: A message-passing interface standard. *Internat. J. Supercomputing Applic.*, 8:169–416, 1994.

[9] C. Koebel, D. Loveman, R. Schreiber, G. Steele Jr., and M. Zosel. *This High Performance Fortran Handbook.* MIT Press, Cambridge, MA, 1994.

[10] MPICH Development Team. Mpich home page, 1993. http://www.mcs.anl.gov/home/lusk/mpich.