CUMULVS: Extending a Generic Steering and Visualization Middleware for Application Fault-Tolerance *

Philip M. Papadopoulos, *phil@msr.epm.ornl.gov*
James Arthur Kohl, *kohl@msr.epm.ornl.gov*
B. David Semeraro, *semeraro@msr.epm.ornl.gov*

Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367

Abstract

CUMULVS is a middleware library that provides application programmers with a simple API for describing viewable and steerable fields in large-scale distributed simulations. These descriptions provide the data type, a logical name of the field/parameter, and the mapping of global indices to local indices (processor and physical storage) for distributed data fields. The CUMULVS infrastructure uses these descriptions to allow an arbitrary number of front-end "viewer" programs to dynamically attach to a running simulation, select one or more fields for visualization, and update steerable variables. (Viewer programs can be built using commercial visualization software such as AVS or custom software based on GUI interface builders like Tcl/Tk.) Although these data field descriptions require a small effort on the part of the application programmer, the payoff is a high degree of flexibility for the infrastructure and end-user. This flexibility has allowed us to extend the infrastructure to include "application-directed" checkpointing, where the application determines the essential state that must be saved for a restart. This has the advantage that checkpoints can be smaller and made portable across heterogeneous architectures using the semantic description information that can be included in the checkpoint file. Because many technical difficulties, such as efficient I/O handling and time-coherency of data, are shared between visualization and checkpointing, it is advantageous to leverage a checkpoint/restart system against a visualization/steering infrastructure. Also, because CUMULVS "understands" parallel data distributions, efficient parallel checkpointing is achievable with a minimal amount of effort on the programmer's part. However, application scientists must still determine what makes up the essential state needed for an application restart and provide the proper logic for restarting from a checkpoint versus normal startup. This paper will outline the structure and communication protocols used by CUMULVS for visualization and steering. We will develop the similarities and differences between user-directed checkpointing and CUMULVS-based visualization. Finally, these concepts will be illustrated using a large synthetic seismic dataset code.

1 Introduction

Scientific simulation programs have evolved from single CPU serial operation to parallel computing on a heterogeneous collection of machines. Many scientists are now comfortable developing PVM- or MPI-based parallel applications for their core computation. However, they are forced to utilize inflexible post-processing techniques for visualizing program data due to a lack of tools that understand the distributed nature of the data fields. Issues such as extracting data that has been distributed across processors and insuring time coherency of a global view hinder the use of on-line visualization and steering. CUMULVS is an infrastructure library that allows these programmers to insert "hooks" that enable real-time visualization of ongoing parallel applications, steer program-specified parameters, and provide application-directed checkpointing and recovery. CUMULVS allows any number of "front-end" visualization tools and/or steering programs to dynamically attach to a running simu-

*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

lation and view some or all of the data fields that a simulation has published. One key to the success of the CUMULVS software is that commercial visualization packages can be used to provide the graphical processing. CUMULVS can be then thought of as a translation layer that accumulates parallel data so that traditional visualization packages can be used for processing. The libraries handle all of the connection protocols, insure consistency of both steered parameters and visualized data across the parallel computation, and recover in the face of network (or program) failure. The fault-tolerant nature of the attachment protocols insure that a running simulation will not hang if an attached "viewer[1]" becomes unresponsive via an (unexpected) exit or network failure.

Viewer programs can abstract the data field and treat it as if it existed in a single flat memory. Figure 1 illustrates this simple but effective abstraction. CUMULVS takes care of the nitty-gritty details of "accumulating" the data from the parallel computation and presenting the data to the viewer as a single array. One issue with this approach is the efficient use of the network connecting the visualization workstation to the parallel application (which may be running on a completely different architecture). Although network speeds are improving, most users still only have 10 megabit Ethernet connections to their workstation. The CUMULVS angle is to allow the viewer to dynamically determine both the extent and the granularity of the data that it wants to see. One can choose to see all of a very large data field at a coarse resolution, or some of the data field at a fine resolution. The downsizing of data is performed in parallel within the simulation and only the data desired is transferred over the "skinny" pipe. Viewers are independent from each other so that different users can see different data fields, or different parts of the same data field, at at the same time.

To this point, little has been said about cost or effort on the part of the application programmer. The CUMULVS library supports C and Fortran 77 interfaces, but models its data decompositions after those presented in HPF [9]. The programmer must describe a data layout (generalized block-cyclic, for example) and a virtual processor array for each data field. The following must be specified to allow the libraries to convert global addresses requested by a viewer to local memory across the parallel application: the global extents of data field, the virtual processor map from logical to physical nodes, and local storage declarations. In general, it takes four subroutine calls to the CUMULVS library to enable visualization: CUMULVS initialization, data field decomposition definition, field definition based on a defined decomposition, and data transfer. The decomposition and field definition subroutines are patterned so that HPF inquiry commands could be used to automatically provide the parameters and greatly simplify the interface. The call to the data transfer subroutine (`stv_sendToFE`)is placed in the body of the main simulation loop. It is in this routine that all viewer connections and parameter updates take place, allowing the programmer to specify a particular point in his/her code when data fields are valid for reading. If no viewers are attached to a running simulation, the overhead to call this routine is negligible and translates to a single message probe. Steerable parameters, defined in a similar manner to fields, are updated in `stv_sendToFE`, which returns the number of steering parameters that were updated during the call. Programs can inquire if a specific parameter changed during the data transfer via a function call.

CUMULVS was designed for on-line visualization and steering of program-specified parameters. However, the large burden on the part of the programmer is to specify the fields and decompositions. Once the definitions have been completed, other "agents" can be used to operate on the data. In particular, programs can specify data fields that should be checkpointed. An external monitoring program can gather checkpoints from an ongoing calculation via the same mechanisms that a viewer uses. Because much is known about the data field (type, dimension, decomposition), these checkpointing agents can provide some unique capabilities beyond those achievable by core-image checkpointing. For example, tasks can be migrated across heterogeneous hosts and restarted using the type and dimension information. Even more interesting is the capability to checkpoint a parallel application running on a particular number of nodes and restart the application on a different number of nodes, and have the data be placed properly in the new decomposition. Also, because the user decides precisely what data CUMULVS needs in its checkpoints, the amount of data collected can be significantly smaller.

In this paper, we will discuss some of the CUMULVS connection and data protocols, how we have extended the visualization library to include user-directed checkpointing, and how these concepts have been put into practical use in a parallel synthetic seismic dataset generation program.

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

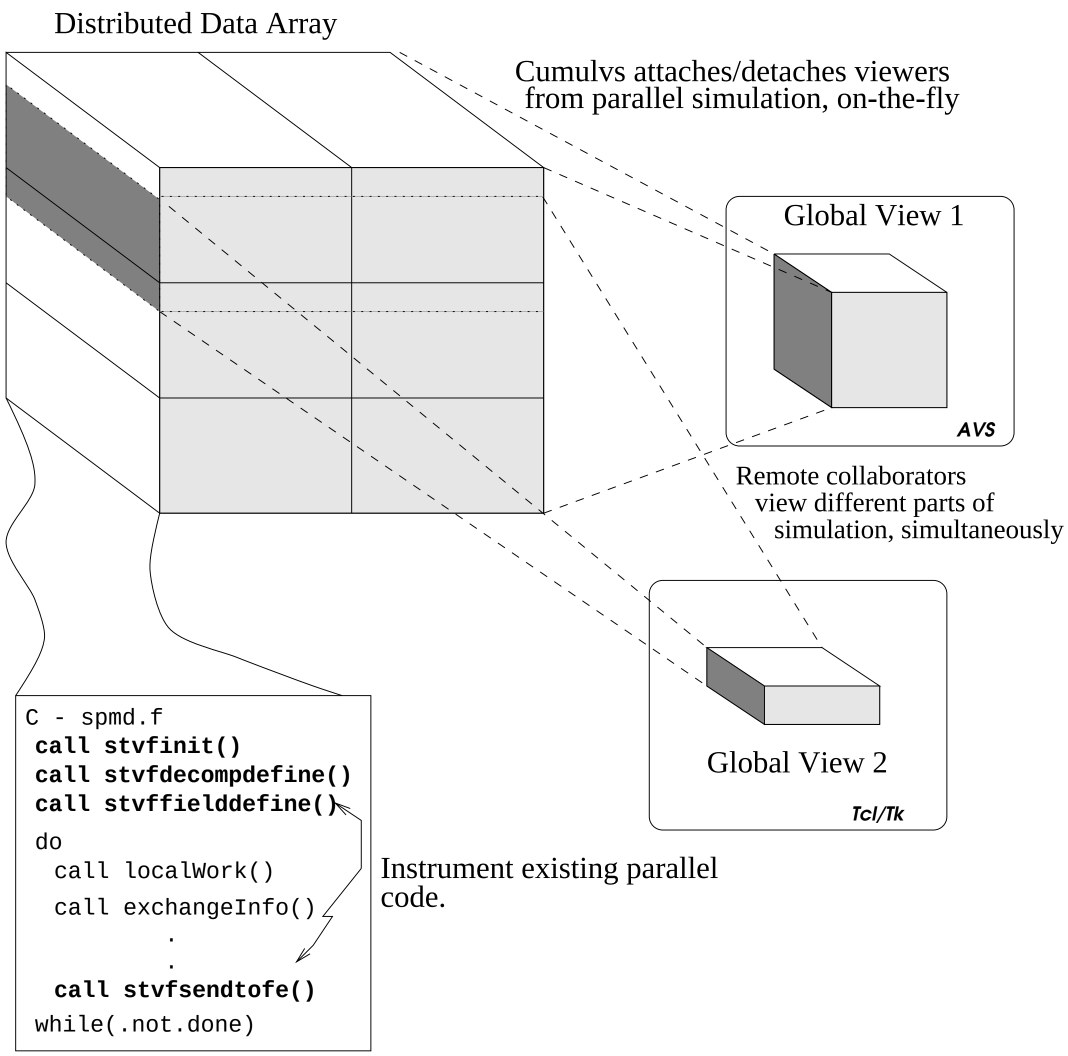

Figure 1: Fundamental abstraction of CUMULVS. Allow multiple viewers to collect distributed data from a running parallel application and present the array as if it were a large homogeneous monolithic dataset.

2 CUMULVS 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 manages all aspects of the dynamic attachment and detachment of viewers to a running simulation. It also provides a method to checkpoint heterogeneous applications, automatically restart an application, and/or bootstrap a checkpointed program. There are several runtime issues for which CUMULVS provides a solution: time-coherency of data extracted from a simulation, guarantees that a steering parameter will be updated at the same logical timestep across a simulation, and consistency of checkpoint data. The libraries do not block or synchronize an application unless absolutely necessary. Instead, the concept of "loose synchronization" is used where a viewer brackets the timesteps and insures that all tasks are on one of the timesteps contained in the bracket. For example, it is possible that tasks A, B, and C are computing at timestep 10, 12, and 11, respectively. Visualization data extracted from the simulation are marked with the timestep for coherent reconstruction at the viewr. However, steering parameter updates must marked with an "apply at" timestamp. Tasks then locally apply the parameter at the correct timestep.

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' 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.

2.1 Attaching to a Running Simulation

Viewers and simulations are independent until attachment is requested by the viewer. There are four distinct phases of attachment: inquiry, request for attachment, data transfer, and detachment. For a viewer–simulation connection to be initiated, some well-known "magic" piece of information must be supplied. CUMULVS uses the application name as supplied in the initialization call defined in the parallel simulation. This name can be completely different from the executable name and usually conveys some meaning to the user. The application name is registered in a database that the indicates how to contact instance 0 of the application (which must always exist).

2.1.1 Inquiry

Once a viewer has successfully looked up an application and determined the message context or tag that CUMULVS should use for communication, it sends an `Init0` message to task 0. Task 0 responds to indicate the total number of tasks in the parallel application; the number, names, types, and decompositions of fields that are defined in task 0; the number, names, and types of steerable parameters defined in task 0; and the current timestep that the task is currently computing. Task 0 is also responsible for forwarding the `Init0` message to all other tasks in the computation as it is presumed to know the total number of tasks that make up the calculation. The remaining tasks respond with their individual field and parameter information directly to the viewer. If any discrepancies occur in the information, such as an inconsistent declaration of a particular field, the inquiry sequence is deemed invalid. It should be noted that a field does not have to exist in all tasks, so that programs that perform virtualization of processes can be supported. At the end of the `Init0` sequence, the viewer knows all of the field names (and decompositions), as well as the steerable parameters that a simulation has published. Each field and parameter is given a string name that the programmer defines to make the actual variable name something more human-understandable.

2.1.2 Field Request

When a CUMULVS viewer desires to view a particular set of fields, 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. The request is really three phase. In the first phase, the viewer sends which fields are required and waits for each task to return the timestep on which they are currently operating, $timestep0$. Tasks will continue to compute until they have reached $timestep0+1$ and then wait for the viewer to provide the next message indicating the timestep to start sending field data. Once the viewer has heard from all tasks, it is able to compute the maximum timestep that any node has achieved. It broadcasts this timestep, $timestep1$, to all tasks. The tasks are then free to compute until they reach $timestep1$, at which point they send the requested data fields to the viewer. This sequencing is critical for parallel programs that synchronize themselves through message passing within an iteration (the typical case). When the field request arrives at a task for a tightly synchronized program, some of the tasks may already be at timestep t, while others may have progressed into timestep $t+1$. If we simply block the tasks when they first process the connection request, the parallel program may freeze. This is because some of the tasks may have "missed" the initial message and gone to the next time step. These tasks may not be able to complete timestep $t+1$ because other tasks may be blocked at timestep t. Hence, to eliminate this race all tasks mark their current timestep and continue on to the next one. The viewer is then able to hear from all tasks in the computation and the possible field request race is eliminated. There are several things to note in the connection protocol. If any of the tasks exit during the three phase startup, then a viewer transmits a 'FieldHalt' so that tasks may break out of any wait loop and abort the field request protocol. If the viewer exits, then each task will abort the field request protocol and continue computing. This keeps the tasks from stalling on what is a recoverable failure.

2.1.3 Data Transfer

Once a valid connection sequence has completed, each task sends its data to the visualization front-end. However, tasks are not allowed to get arbitrarily far ahead of a viewer. Instead, flow control is used so that if the front-end is having trouble keeping up with the simulation, it will effectively slow the calculation. Viewers may choose to retrieve information at any frequency to alleviate this slowdown. In this case the the simulation sends the requested data to the calculation every *pth* timestep for a visualization frequency of p.

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.

2.1.4 Shutdown

Tasks understand two types of disconnects: one is to quit sending data for a particular field to a viewer (`FieldHalt`), the other is to discontinue sending all data to a viewer (`FieldHaltAll`). When a viewer first requests a connection, each task posts an exit notification message for the attaching viewer. If the viewer unexpectedly exits, then the notification in effect generates a `FieldHaltAll` message to the task. Hence at *any* point in the connection or data transfer sequences, checks are made in the library for halting messages. On reception of these halting messages, whether generated by the viewer or by a notify, a task will no longer block waiting for communication from the viewer. This keeps the parallel application robust to viewer failure.

2.2 Coordinated Computational Steering

CUMULVS supports coordinated computational steering of applications by multiple collaborators. A token scheme prevents conflicting adjustments to the same steering parameter by different users and 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.

With the exception of the token locking, requests for steering use the same synchronization mechanisms as viewers so that a "steerer"/viewer can determine the exact timestep that a simulation is on. Also, tasks send an empty visualization field to the viewer if that viewer is only performing steering. In this way, the steerer has precise knowledge of the current timestep of *tasks* in the distributed computation and can tag a parameter update with a specific timestep for its application. Logic in the steerer sets this steering timestep at the earliest coherent timestamp, given the current state of the simulation.

2.3 General Infrastructure Issues

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, VTK, 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.

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.

3 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 2: 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 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()` 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.

4 Fault-tolerance Design

At first blush, it may seem counter-intuitive to logically link checkpointing with steering and visualization. However, in the CUMULVS approach where a small amount of effort is asked of the programmer to describe data distributions, a windfall of opportunities arise, checkpointing being only one of these. The data descriptions coupled with a method of dynamically attaching to an ongoing code leads to a variety of scenarios. If, instead of a viewer or steerer, ones considers that any general "agent" may use the CUMULVS connection protocols, then the door is opened for other agents to enter a parallel simulation and extract information from that simulation. On the other hand, steering agents can affect the ongoing calculation, thus allowing a control loop to effectively be closed. In this section, we consider the experimental checkpointing agent that we have implemented using CUMULVS concepts. The basic idea behind our approach is that the program can direct when a checkpoint should occur and what essential data is needed to restart. An external agent can handle all of the data extraction and logic to commit a checkpoint and to restart a fully or partially failed application.

In CUMULVS, 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.

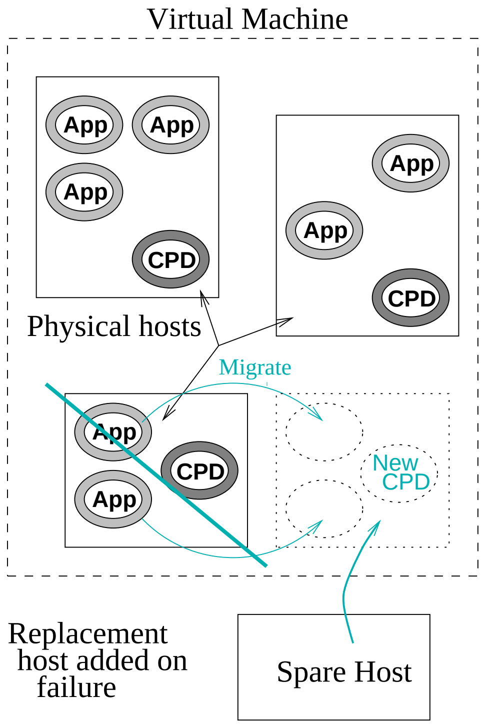

Figure 3: Checkpointing daemons (cpd's) make up a parallel fault-tolerant program that monitors a user's parallel application for failures. Cpd's also add spare hosts to the virtual machine and manage task migration to the new host

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. With the program-provided descriptions, 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.

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. Figure 3 illustrates the the basic design of the cpd's. 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, with viewer-style flow control being employed by the checkpointing daemon. 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.

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 in addition to the time take to save and replicate checkpoint data. Currently, tasks pack and send checkpoint data to the local cpd, which saves the data on behalf of all tasks. This method is too slow for large scale practical application and will be replaced. The new scheme will employ the cpd as a coordination mechanism and tasks will write their own checkpoint data. This new scheme will allow the use of parallel file I/O on systems that support it.

4.4 The Next Steps for Checkpointing

The *cpd* makes up a parallel application that provides its own fault-tolerance. In essence, processes on one host contact only one node of the parallel cpd program. The makes the underlying CUMULVS assumption of gathering to a serial computation still valid. A powerful generalization would be to allow parallel programs to connect to other parallel programs. CUMULVS-style connection protocols where the underlying library handles all of the details will allow others to produce parallel-to-parallel steering, visualization, coupling, checkpointing, or some other type of interaction agent. One important issue for this to be a success is to implement efficient routines to perform redistribution of data. For example, a simulation may store a data field in a block-cyclic distribution across 16 processors while a parallel visualization program may desire part of this data in a 4 processor block distribution. It will also take significant analysis to design connection protocols that are reliable and recoverable like the current connection protocols. This type of interconnectivity would open the doors to a large number of new coupled applications.

5 Seismic Code

The seismic code used as an example in this paper simulates the propagation of an acoustic signal through a heterogeneous media by solving the scalar wave equation,

$$\frac{1}{c^2}\frac{\partial^2 u}{\partial t^2} - \Delta u = f(x,t).$$

Here, $c(x)$ represents the local velocity of acoustic waves, $u(x,t)$ is the pressure field, and $f(x,t)$ is the source term. This simulation has been used to create a synthetic seismic dataset that will eventually be used to calibrate seismic analysis codes. The simulation is a finite difference approximation to the three-dimensional wave equation. Second-order centered differences are used to discretize the time terms. Tenth-order centered differences are used for the spatial term. Mesh spacing is uniform in all three dimensions. The computational mesh is regular and Cartesian.

The synthetic seismic dataset project simulates a seismic survey on computers. The survey is done by simulating many thousands of events called "shots." Each shot consists of a signal generated at a particular point in the domain, the propagation of sound waves in the media, and the collection of time history data at an array of receivers in the domain. This is analogous to the physical case where data is collected in the field. The data for each shot is the acoustic pressure collected at thousands of receivers, hundreds of times a second. This results in a large amount of output data. This volume of data is then multiplied by the thousands of shots required to define a geology. The entire project represents too much computation to be done by one entity and is in fact a cooperative project involving several national labs and industrial partners.

CUMULVS was used to attach to the simulation and extract the sound 'pressure' field. The code was modified to allow the arbitrary placement of "thumps" within the computational grid. A thump represents a point source of sound energy and in actual field surveys is usually generated by an explosive charge set in a mechanical device that impacts the ground to create the sound source. Furthermore, checkpointing was put in place to provide for fault tolerance. The following section gives our empirical observations about the usability and programmability of CUMULVS from the user's perspective.

5.1 Programmability and Usability

The additional instrumentation needed for CUMULVS visualization was really quite modest. Approximately 30 lines of code was added to the existing parallel program, which was written in FORTRAN. The bulk of the added code was in terms of describing the data layout of the various fields. Some small amount of additional logic was added so that new seismic thumps could be set off interactively. CUMULVS steering allowed us to insert thumps anywhere in the three-dimensional domain – something that is possible but expensive in the field because of the drilling costs incurred for placing a charge. The interactive feel of the simulation was governed by the speed of the computation and not so much by overhead costs in CUMULVS itself. One noticeable degradation appeared when trying to extract data from a simulation running on an Intel Paragon. This was due to poor TCP/IP connectivity off of the Paragon compute partition and should be regarded as an inherent problem with Paragons.

However, when no viewers are attached, the overhead is immeasurable in terms of overall program speed. Several measurements were made with and without CUMULVS instrumentation with no observable difference in run times.

Instrumenting the code for fault-tolerance was quite a bit more challenging. The recovery modes were such that live nodes had to react to dead nodes and restart. The inherent problem is that tasks may block waiting for a message from a dead node. The message passing routines (which were encapsulated in a single file) had to now support error-return semantics when a dead node was discovered. The other option, error-exit semantics, would have meant that on an error, nodes would simply call `exit()` and the CUMULVS checkpointing daemons would be responsible for restarting a complete application, rather than just replacing failed nodes. To support the error-return semantics, messages were wrapped so that an error notification would cause a blocking receive to return with a failure message. This wrapping was straightforward because the CUMULVS internals use a similar scheme for handling failures and the logic was already written. The more difficult part was to adjust the logic in the program to handle starting normally versus starting from a checkpoint. While the error logic and deciding when to checkpoint had to be added to the seismic code, the effort was not onerous. It took about a days worth of work to change our statically configured parallel code into a fault-tolerant application with checkpointing.

Checkpointing overhead is a serious concern. Since the checkpointing is in a preliminary stage, we did not rigorously characterize the overhead of checkpointing for the seismic code. Instead, we found that checkpointing every 20 iterations seemed to have light impact on a small network of workstations. The current CUMULVS implementation of full checkpoint replication is too costly and plans are underway to allow users parameterize the amount of replication needed by a particular application.

6 Conclusions

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.

The current viewer library provided in CUMULVS assumes that the viewer programs themselves are serial. A useful generalization would be to allow connections of parallel visualization agents. A parallel-to-parallel scheme would certainly require a library of transformation methods to redistribute data from one decomposition to another. This would certainly require substantial protocol changes to produce an efficient, robust and user-friendly system.

The experimental checkpointing works, but the checkpointing code has evolved over time and has become increasingly difficult to reason with to eliminate race conditions within the checkpointing daemon. Much of the code probably needs rewriting to make the daemon as robust as possible to failures.

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.