Efficient and Flexible Fault Tolerance and Migration of Scientific Simulations Using CUMULVS

James Arthur Kohl *kohl@msr.epm.ornl.gov* *
Philip M. Papadopoulos *phil@msr.epm.ornl.gov*
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN 37831-6367

Abstract

Many practical scientific computer applications would benefit from a simple checkpointing mechanism that provides automatic restart or recovery in response to faults and failures, and enables dynamic load balancing and improved resource utilization using task migration. However, developing applications with such capabilities, especially in distributed, heterogeneous operating environments, is very challenging. CUMULVS is a middleware infrastructure for interacting with parallel scientific simulation programs and supports online visualization and computational steering. Using semantic information provided by user-level specifications of selected program variables, CUMULVS interprets distributed data decompositions across heterogeneous collections of computing resources. It extracts and assembles subsets of local decomposed application data to form global views of the data. The base CUMULVS system has been extended to provide a user-level mechanism that assists in the collection of checkpoints for parallel simulations or other calculations. Via the same semantic interface used to identify and describe data fields for visualization and parameters for steering, the user application selects the minimal program state necessary to restart or migrate an application task. The CUMULVS run-time system utilizes this information to efficiently recover fault-tolerant applications by restarting failed tasks. Application tasks can also be migrated – even across heterogeneous architecture boundaries – to achieve load balancing or to improve a task's locality with a required resource. CUMULVS handles the tedious and error-prone tasks involved, leaving the developer of fault-tolerant or migrating applications to focus on the application-specific design details. This paper describes the CUMULVS interface for checkpointing, the issues faced in utilizing this interface when developing fault-tolerant and migrating applications, and the direction of future research in this area.

*Research supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, U. S. Department of Energy, under contract No. DE-AC05-96OR22464 with Lockheed Martin Energy Research Corporation.

1 Introduction

Next-generation scientific software applications will be significantly more dynamic and complex than today's traditional, statically configured programs. High levels of interactivity, interoperability, fault-tolerance and mobility will be required to fit into wide-area collaboration and large-scale computational grid resources. Two key enabling technologies required for this flexibility are the capability of scientific applications to present a semantic and functional interface to their internal information and services, respectively, and migration from one computational resource to another.

The CUMULVS project [1, 2, 3, 4] takes an important step towards this goal by providing a middleware infrastructure for interacting with an ongoing parallel computation through online visualization and computational steering. Within this scope, CUMULVS provides protocols and APIs that allow an independent "viewer" application to dynamically attach to, interact with and detach from a running calculation or simulation. Using CUMULVS user-level library calls (in C/C++ or Fortran), the application describes its data fields and their decompositions (if any) across the parallel application tasks and identifies any steerable parameters. This semantic interface is sufficient to support transparent, interactive connections to the application, potentially across heterogeneous architecture boundaries, for extracting desired visualization data, and for manipulating the values of algorithmic or model parameters.

CUMULVS interprets the user-supplied information on distributed data decompositions to extract and assemble global views of decomposed application data. Unlike systems such as DICE [5], where whole copies of each data field are placed in a globally shared file structure using DDD [6] and HDF [7], in CUMULVS the data movement is demand-driven and the viewers dynamically extract only requested subregions of data fields from each application task. This reduces the application overhead in most cases and provides more flexible multi-viewer collaboration scenarios. CUMULVS has been integrated with parallel applications written using PVM [8], MPI [9] and InDEPS [10], and can be applied to applications with other arbitrary communication substrates.

The simple model of interaction in CUMULVS can be generalized and extended to encompass more elaborate functionality. CUMULVS can be applied to assist in the development of applications that support automatic fault recovery and on-the-fly migration. Fault recovery is clearly an important feature for any long-running application. An automatic response, that recovers or restarts failed tasks, is also essential to keep the application running indefinitely without continuous user monitoring. Task migration is useful for load balancing and resource utilization. Often execution time would decrease if tasks could be efficiently relocated to less-loaded resources. Moving tasks closer to necessary resources also improves locality and decreases communication overheads, thereby boosting resource utilization and efficiency.

1.1 CUMULVS Checkpointing Overview

The CUMULVS checkpointing facility capitalizes on the semantic interface used to identify and describe data fields for visualization and parameters for steering. These same data descriptions can be marked by the user to identify which elements contribute to the minimal program state, as needed for recovering or restarting each given task. The application can periodically direct CUMULVS to dump the selected program variables and save them as a checkpoint. Then, when a task fails it can be restarted, using this checkpoint data to reset its program state. In conjunction with the recovery of failed tasks, any other cooperating tasks can either be restarted or rolled back to the same saved state using their checkpoint data, so the application as a whole can continue from a common point. Any computation that occurred since the last checkpoint will be lost, therefore the frequency of checkpoint collection requested by the application is often adjusted to hedge against failures, and balance this loss against the periodic checkpoint overhead.

CUMULVS' user-level approach to checkpointing is considered "non-transparent" because it requires the programmer to modify the application source to coordinate the checkpointing activity [11]. In this case, the application defines the point(s) in its computation where checkpoints can be "consistently" collected, across concurrent sets of tasks, and which data should be included in checkpoints. Consistency here relates to identifying a global state for all tasks such that the computation can continue without error if recovered to this state (see below). It is the application's responsibility to coordinate the consistency of the saved program state so that all tasks can be correctly reset to some uniform state in the event of a failure. The application as a whole must correctly restart and continue on from this state, as if the application had never failed.

Creating these checkpointing specifications, especially determining the consistent global state, can be quite challenging. However the benefit gained for this effort is that CUMULVS can oversee the tedious and error-prone tasks involved in actually checkpointing the application. CUMULVS extracts the checkpoint data from each task and assembles it into globally committed checkpoints. It handles the automatic fault-notification and recovery procedures, to spawn replacements for failed application tasks and direct the remaining tasks to be restarted. CUMULVS determines the last completed checkpoint, sends it to the application tasks, and loads the checkpoint data into the program variables.

To accomplish all of this, the application must select the necessary program state to save, and the proper point in each application task to save the checkpoint data, so that the overall global state can be reproduced to some degree of synchronization. The consistent global state must include the program state of each task, any external (file) state, and the state of the communication channels among the tasks. Because CUMULVS is a user-level system that does not control operating system state, the communication substrate can be considered to be memoryless. Therefore, it is sufficient to insure that all tasks' states are checkpointed either after all sent messages have been received, or equivalently before any new pending messages are sent.

Typically, this is a straightforward task in iterative computations where there is a distinct logical "end" to each iterative stage, at which point all tasks can assume an appropriate synchronization state. In this case there is already a loose synchrony among the iterating tasks, implicit in their dependence on messages from cooperating tasks. If a task were to fail before receiving one or more messages for a given iteration, then the application could be rolled back to the end of the previous completed iteration. Any messages that were sent but not received would be re-sent as the tasks re-execute the failed iteration.

For applications which do not fit algorithmically into an iterative model, the communication must be synchronized before checkpoint data can be consistently collected. This could be done periodically using a variety of distributed algorithms, such as that presented by Chandy and Lamport that uses "message markers" to indicate a locally consistent task state [12]. It is the application programmer's responsibility to determine and implement the proper synchronization algorithm for this stabilization of an application's global state.

In all cases, any external state implicit in file pointers or other system services must be manually reproduced by the application. CUMULVS can assist in this task by saving and restoring any user-defined variables that mimic or record the external state. But, in the current prototype implementation, it is up to the application to re-open files and seek to the correct location, etc.

In addition to fault tolerance, CUMULVS checkpoint data can be used to migrate tasks to different computational resources. This is accomplished by collecting a checkpoint (or deferring to the last saved checkpoint), killing off the tasks to be migrated, moving the checkpoint data to the new resource(s), and then restarting the migrated tasks using the checkpoint. There is little functional difference between this operation and standard failure recovery. Any subset of the application tasks might be migrated at one time, including the entire set.

1.2 Benefits of Approach

Because the checkpoints in CUMULVS contain user-supplied semantic information, including the name, type, storage allocation and any decomposition of data fields, it is possible to translate checkpoint data from one system architecture's format to another. This means that a task can be checkpointed on one system and then restarted or migrated heterogeneously to a different architecture. Also, because the user application has selected only the relevant portions of the computational stack for checkpointing, the resulting CUMULVS checkpoints can be significantly smaller than an entire core image. This can increase the efficiency of restarting a task after a failure, and can make load balancing via task migration a more feasible option. The current release of CUMULVS checkpointing has already been used to demonstrate on-the-fly cross-platform fault-tolerance and migration of several production parallel codes.

CUMULVS is not intended for fine-grained checkpointing, at the level of individual program statements or messages. By only saving checkpoint data periodically (at the user's discretion), CUMULVS is an appropriate choice for most scientific applications where the desire is typically to limit the loss of computing cycles in response to failures. CUMULVS checkpointing is often done at a coarse level, e.g. every so many iterations in the main computational loop, or between the high-level phases of a multi-phase computation.

The remainder of this paper compares CUMULVS to existing checkpointing systems, describes the details of the CUMULVS interface for checkpointing, and further explores some of the issues faced when utilizing this interface to actually develop a fault-tolerant or migrating application. Specific plans for future research in this area will also be discussed.

2 Background

The CUMULVS approach to checkpointing has several advantages over traditional core-image checkpointing. Many transparent checkpointing environments, such as CoCheck [13], MPVM/MIST [14, 15], CLIP [16], Fail-Safe PVM [17], Isis [18], Totem [19], Condor [20] and others [21], are designed for single architecture programs. CoCheck works with PVM and MPI to save the entire binary image of a program and move it to another similar machine. While this system works well for fault recovery, the size of these binary dumps make CoCheck impractical for migrating tasks to achieve medium- or fine-grained load balancing, and the binary nature of the data prevents movement across architecture boundaries. MPVM and MIST have similar capabilities for PVM-based applications. CLIP performs special-purpose semi-transparent checkpointing on parallel applications written for Intel Paragons. Fail-Safe PVM uses a global synchronization to coordinate consistent checkpoints in a parallel application, but saves checkpoint data transparently in architecture-specific formats, precluding heterogeneous restart or migration. Isis and Totem use the concept of "virtual synchrony" to greatly simplify the logic of writing fault-tolerant programs, but require either a partial or total ordering of all messages in the parallel program. While this ordering supports a very fine granularity of checkpointing, at the level of individual message synchronizations or transactions, it carries a high overhead and is impractical for applications that execute on large numbers of nodes. Condor, a distributed batch processing system that schedules jobs for execution on UNIX systems, can arbitrarily checkpoint single processes and migrate them to other machines, but does not support groups of tasks that communicate.

These transparent checkpointing systems explore a wealth of different checkpointing issues, especially those related to parallel or distributed applications and environments, but they all limit their focus to checkpoints that can be automatically extracted and used for restart or migration on systems of the same architecture. To allow migration or restart across architecture boundaries, or other powerful capabilities, the checkpointing system must have additional information from the user application to semantically identify the program state data. Several "non-transparent" checkpointing systems exist that provide user constructs and libraries for user-directed checkpointing, including Dome [22], CHIMP/MPI [23], Calypso [24], Libft [25] and COSMOS [26]. The Dome system provides a mechanism to instrument C++ applications, written on top of PVM, for checkpointing and heterogeneous migration. Using user-inserted checkpointing directives and special program variable declarations, potentially with the assistance of compiler preprocessing, C++ objects in Dome can be checkpointed and migrated to heterogeneous resources. CHIMP instruments the MPI message-passing library for fault-tolerance. Calypso supports its own source language (CSL) that extends C++ to create a powerful, fault-tolerant software system. Libft is a part of the Software-implemented Fault Tolerance (SwiFT) system, and is a C library that provides a variety of programming constructs for fault tolerance and recovery mechanisms. COSMOS is a special-purpose operating system for spacecraft that provides software-implemented fault-tolerance on distributed memory multiprocessors for long-life interplanetary missions.

While all of these systems provide useful and elaborate checkpointing capabilities for fault-tolerance and migration, each relies on a specific set of languages or operating environments. The CUMULVS checkpointing facility is a simple prototype, perhaps without some of the more elegant functionality and optimizations of these other systems, but is generally applicable to applications developed using a variety of programming languages, message-passing and communication systems, and operating environments. While CUMULVS requires certain features from its underlying message-passing substrate (for communication between CUMULVS daemons and the application tasks), there are no restrictions on the nature of the communication systems or languages which the application itself can utilize internally.

3 CUMULVS Checkpointing

The fundamental premise in CUMULVS checkpointing is that the application program can best direct when checkpointing should occur and what essential data is needed for restart and migration. The user can indicate the proper point within each application task for collecting checkpoint data, so that the resulting checkpoints represent consistent states across the entire application. The application can direct how often the state needs to be saved, thereby controlling how much overhead is incurred and how much computation is lost when a task restarts. This so-called *user-directed* checkpointing requires some work by the programmer, but is essential to provide the additional efficiency and flexibility in CUMULVS. The library infrastructure can handle the low-level details, by coordinating the data extraction from within application tasks, and by processing the logic in the run-time system to organize and "commit" consistent checkpoints from the individual task data. Committing a checkpoint involves synchronizing the CUMULVS run-time system to insure that all tasks have submitted their data for a particular checkpoint. Given the user's checkpointing specification, CUMULVS can do the actual work in restarting or migrating application tasks.

There is, of course, a price to pay for the automatic handling that CUMULVS provides. The user must instrument the application to include some semantic details, and the program initialization must be modified to optionally allow a restart from a checkpoint. The amount of additional code required for this is not immense – typically on the order of tens of lines of code – and requires mostly knowledge of data field decompositions and the high level program initialization which, hopefully, should already be well understood. Generally speaking, if the user requires the fault-tolerance and/or migration capabilities then the effort involved is easily justified, especially if the application in question has already been partially instrumented for visualization and steering using CUMULVS. Nonetheless, making any application fault tolerant is a complex and challenging ordeal. CUMULVS strives to ease this task by transparently and automatically handling many of the arcane portions. This leaves the application developer to focus on the more important aspects of selecting the program state and coordinating the consistency of checkpoint submissions. In any case, it is hoped that as part of future CUMULVS research some assistance may be provided to alleviate the burden of application instrumentation, perhaps in the form of a graphical user interface (GUI) or CUMULVS compiler directives and preprocessing (see [27, 28]).

To enable automatic checkpoint handling in CUMULVS, the programmer must specify what variables need to be saved, and must provide an alternate program initialization for restarting from a checkpoint. During such a restart, CUMULVS commits and retrieves the most recent coherent checkpoint, and then sends each application task its portion of the checkpoint data and loads this data into the user's variables. For this to be possible, CUMULVS must be told the location in memory and the storage size of all checkpointed variables. This information is provided by the user via semantic data field declarations in each application task. The user application must also, if so instructed, pass over the default initialization of these program variables and instead allow CUMULVS to set them to the desired restart state.

With respect to efficiency, CUMULVS operates under the assumption that fault recovery and task migration are sparsely applied operations, and so design choices have been made to minimize steady state overhead. While checkpointing in any system is relatively time consuming, care is taken in CUMULVS to avoid undue perturbation to the application. It is assumed that machines are generally fairly stable and that a program should only suffer significant overhead when there is an actual failure or migration. Each task executes with only the synchrony required by the user application, and each task checkpoints its data independently. The interactions involved in collecting checkpoints from application tasks are not themselves synchronizing operations, but rather are controlled by a simple flow control protocol. Tasks are not held back unless the run-time system is unable to keep up with the rate of application checkpoint data submissions. A full checkpoint commitment synchronization is not done unless explicitly required for a migration or fault recovery. Otherwise, such commitment is done in a "lazy" manner, as messages flow through the run-time system to propagate notification or replication of the latest checkpoint submissions. When all tasks' submissions for a particular checkpoint are globally known to have completed, the checkpoint may be considered committed.

The following two subsections describe more details of the CUMULVS checkpointing run-time system and its design issues, respectively.

3.1 Run-Time System Architecture

In CUMULVS, much of the logic needed to reliably and correctly restart failed parallel application tasks has been moved to a separate process called a "checkpointing daemon" (CPD). The current CUMULVS design has a separate checkpointing daemon on each machine where application tasks are running. Figure 1 illustrates the basic organization of the CPDs. The set of daemons works together as a separate, dynamic, fault-tolerant program, independent from any user's code. CPDs monitor the user application and its computational resources for failures, and oversee any checkpointing, restarting or migrating of application tasks. CPDs are also responsible for adding spare hosts (where possible) in response to resource failures.

From an application's perspective, the CPD provides two

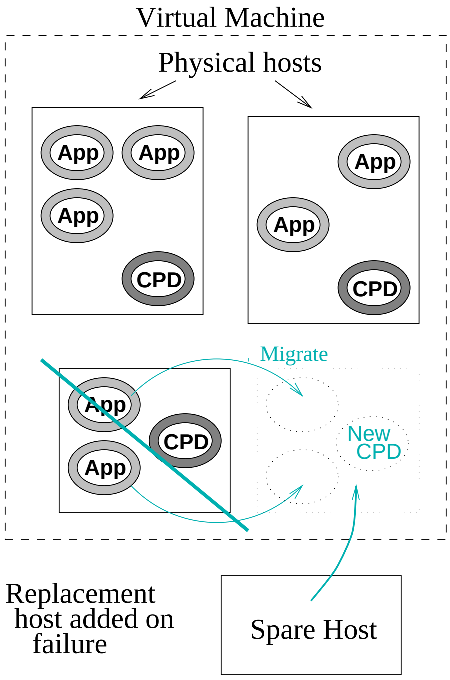

Figure 1: Checkpointing Daemon (CPD) Organization

basic checkpointing functions, saving a checkpoint for the application and loading a checkpoint for restart. Logically, however, the CPDs can be utilized in several modes. They can serve as a background run-time monitor to keep the application running without user intervention. Instead of the user periodically checking to see that the application is proceeding, the CPDs can continuously oversee the tasks using fault notification features of the underlying messaging system. Any failed tasks are restarted automatically, in coordination with restarting or rolling back any cooperating tasks, even if the user is not present. Given a user-supplied list of alternate computing resources, the CPDs can even add new hosts when others crash, subsequently replacing any tasks that were executing on the failed hosts.

The user can invoke a CPD "console" at any time to manually direct the CPD group in restarting or migrating a subset of application tasks, for improved load balance or better resource utilization. The CPDs will determine the most recently saved complete checkpoint, terminate the tasks which are to be migrated and then restart the given tasks on their new resources. Any cooperating tasks are also rolled back or restarted using the same checkpoint. At present there is no built-in CUMULVS mechanism for automatic load balancing. However, the user can, with some effort, construct their own CPD console for automatically controlling CPD migration operations using various load balancing algorithms.

Additional operations can be initiated using the default CPD console, such as restarting an entire application, as saved in some earlier checkpoint, on an arbitrary set of new resources. This capability raises many interesting possibilities, especially if the new resources are of a different system architecture or utilize a different number of compute nodes. Using the data field description information in the checkpoint data, it is straightforward for the CPDs to restart the application on a different architecture by translating the data format of the checkpoint (typically done using

XDR and the underlying message-passing system). However, many issues arise when the number of available computational nodes changes and the decompositions of the data fields must be redistributed. Such elaborate reconfigurations of applications using CUMULVS checkpoints have been explored in [29], and to date several on-the-fly reconfiguration experiments have been successfully applied using both toy and production parallel simulations.

3.2 Run-Time System Issues

The most time-critical operation in CUMULVS checkpointing is collecting the checkpoint data. The CPDs use an asynchronous scheme, such that each task sends its checkpoint data when the user code makes a call to `stv_checkpoint()` (see Section 4.2). The application code does not explicitly synchronize at each checkpoint. However, a task will block on the subsequent `stv_checkpoint()` invocation if the previous checkpoint has not been received and fully processed by the checkpointing daemon. A simple flow control protocol is employed to release the application task when the previous checkpoint is finished. When the CPD completes the checkpoint processing it immediately sends an "XON" release message to the application task. If the release is received in time, then the task will not wait before proceeding.

It is the responsibility of the CPDs to make sure that a parallel task is restarted from a *coherent* checkpoint that corresponds to the same logical time step, or point in the computation, for all tasks. Because application programs are not explicitly synchronized, it is possible for the most recent checkpoint to be incomplete, missing one or more tasks' checkpoint data for that invocation or *epoch*. If a failure occurs while in this state, then the CPDs must collectively revert to the last complete checkpoint. This agreement, or *commitment*, to a particular checkpoint epoch is done only as needed during a restart or migration, to avoid unnecessary overhead or synchronization of the CPDs while collecting checkpoint data. The CPDs can also commit a checkpoint periodically to free up file system resources by deleting old checkpoints.

The predominant overhead in checkpointing is the time taken to write data to disk or other non-volatile storage. If replication of checkpoint data is desired to allow recovery from multiple simultaneous failures within the system, then inter-machine network bandwidth is consumed to copy data from one machine to another and verify consistency. The CPDs therefore can also impose a small additional computational overhead to coordinate and synchronize this data replication. To reduce this overhead the CPDs are organized in a logical ring, where each CPD need only coordinate replicated data from several "buddy" CPDs. For example, if the user requests a level of redundancy of 2, such that it can recover from 2 simultaneous failures, then 3 copies of each task's checkpoint data are needed at any given time. In this case each CPD would coordinate with 2 other CPDs to insure replication of a task's checkpoint.

An important issue is the level of data replication that should be supported in checkpoints. In the case of small checkpoint files 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 need be retrievable to restart the entire program. On the other hand, if the amount of checkpoint data or the number of application tasks is very large, then full checkpoint replication, especially using standard low-speed networks, is clearly impractical. For this reason, the coordination protocols of the CPDs in CUMULVS have been generalized to support a spectrum of redundancy options.

Another issue relates to scalability and I/O. In the current version, tasks pack and send checkpoint data to the local CPD in messages, and the CPD saves the data on behalf of the tasks. While this communication time can be significantly less than the corresponding file system access time for each task, this method is too slow for large-scale applications. The CPD can quickly become a serial bottleneck if it must manage too many tasks with checkpoint data of sufficient size. An alternate scheme employs the CPD as a coordination mechanism only, and tasks write their own checkpoint data to their local filesystem. This new scheme will allow the use of parallel file I/O on systems that support it.

4 Application Interface

To use CUMULVS for checkpointing, the user must manually specify information about each relevant variable, including an appropriate reference name, the data storage allocation details, the data type, and any distributed data decomposition. Ultimately, using CUMULVS checkpointing should be more automatic by integrating it with a parallel development environment or with compiler cooperation. It is certainly feasible and desirable to build CUMULVS on such systems. Yet the current reality is that the user must use a traditional library interface to define data field semantics and decompositions, and to identify essential program state for restart.

There are several alternatives to this manual annotation, from elaborate graphical user interfaces (GUIs) that guide and assist users through the process, to simple compiler directives to incorporate the missing information. Everything except the data decomposition *could* be made available by conventional compilers, and in some systems like HPF [30] even decomposition information could be extracted automatically. But until such interfaces are developed, the user must make several CUMULVS library calls to describe the internal application structure.

There is some benefit to this direct instrumentation approach versus automatic techniques, in that *subsets* of desired variables can be described for CUMULVS. In automatic instrumentation potentially *all* variables would be included, and it could be hard to exclude loop indices and other temporary storage. If, however, there are hundreds or even thousands of variables to annotate, as is true for some production software, then an automatic system would not only be beneficial but strictly necessary.

The CUMULVS library for defining data field semantics uses HPF-like semantics, such as standard Block-Cyclic, to define data decompositions. These decompositions are subsequently used to define individual data fields. The same decomposition can be reused for multiple data field declarations, and each field can map the decomposition to a distinct logical processor organization. For example, if several data arrays of different sizes were decomposed onto various subgroups of processors, but all were of the same dimension and had the same Block-Cyclic structure, then a single CUMULVS decomposition could be used to describe all the arrays. The only difference from array to array would be the data type, and the number and logical topology of the processors to which it was assigned. In addition to this semantic information for the data fields, each data field definition can be flagged to have that variable included in the saved checkpoint state. Other scalar parameters can also be defined and flagged in this way, if they contribute a portion of the program state.

Besides selecting the essential program variables, the user must also make some changes to the program flow. The more trivial portion of this is merely inserting the single library call needed to request the saving of a checkpoint, `stv_checkpoint()`. This one call initiates the extraction of all user-specified variables, as flagged for checkpointing, in the local task, and packs them into a message for the local CPD. The local CPD in turn coordinates with any remote CPDs to commit a full checkpoint for the application, as formed by the union of each tasks' individual checkpoint data for the same "epoch" or logical time step.

The call to `stv_checkpoint()` within each local task must be made periodically when all data is in a "consistent" state. As discussed in Section 1.1, a consistent state is one such that the global state is known and reproducible, including program variables in the task, messages in the communication substrate and any other external state, such as in files. For CUMULVS checkpointing, the underlying message state is inaccessible except from the task endpoints, therefore it is sufficient to save the local checkpoint either before or after a particular "well-known" message is sent or received in the algorithm. Again, this is all at the user's discretion.

In most iterative scientific computations there is a natural choice for this message, either at the end or the beginning of the main computational loop. Typically, at some point in iterative algorithms some data is exchanged, either in a nearest-neighbor or broadcast communication, to coordinate (synchronize) the computational state across the parallel tasks. The checkpoint data, and therefore the global state, can reliably be saved either before or after this set of messages is sent and received. If the checkpoint is completed (committed) before a failure occurs, then all tasks can be restarted in unison before or after this synchronization phase, and no messages will be left "dangling." Similarly, if a task failed before submitting its checkpoint data, then the committed checkpoint from the previous iteration could be used for consistent recovery. If, however, two tasks were to save their local checkpoint data *in the middle* of this type of message synchronization, or anywhere else where one task had sent a message but the other had yet to receive it, then the state implied by the pending message would be lost, and the message would not be correctly reproduced if the sending task were to restart from the checkpoint.

In applications with non-iterative algorithms there may not be an obvious logical "stopping place" where the message state is reliably known. In this case, the application must force a synchronization of some sort to guarantee the global messaging state before checkpointing any tasks locally. Alternately, for fully independent (embarrassingly) parallel algorithms, where no synchronization of any kind is required among tasks, determining the global state is trivial, and checkpoints can be collected at arbitrary points among the different tasks.

Other than the message state, any other external state must also be identified and saved, including files or services on which the application tasks depend. CUMULVS could eventually provide better support for certain aspects of this external state, such as handling open file pointers if the user specified the proper variables and corresponding file name, etc. For now, however, CUMULVS does not provide any extra support for saving this external state. The best that can be done is to define simple user variables that hold, for example, the file name and current file pointer location, and let CUMULVS reproduce those values in recovered tasks. It is then up to the application itself to actually apply the reconstruction of state, e.g. reopen the file and seek to the correct location.

Perhaps the most invasive part of the user instrumentation involves modifying the control flow of the application for recovery from a checkpoint. When a failed task is restarted, it must be able to accept the loading of checkpoint data to set its starting program state. The application must circumvent the default variable initialization for those variables that are to be set from the checkpoint data. The return value of the call to `stv_cpInit()` that sets up CUMULVS checkpointing (see below) will inform the user of the restart case. Based on that return value, the application task must select between initializing its program state as usual or using an alternate checkpoint restart initialization.

Typically, the default code or routine for initializing the program state can be conditionally swapped out in favor of a call to the `stv_loadFromCP()` routine, which will fill in the program variables with checkpoint data. If the default initialization requires the allocation of dynamic storage for these variables, then that allocation must be separated out from the rest of the initialization so it can be executed for both the default and restart cases. In the worst case, the allocation or initialization of one variable could depend directly on the initial value of another. In this case, some variables may need to be selectively initialized, by interspersing several calls to `stv_loadFromCP()` along with the library calls for CUMULVS data field declarations, to incrementally bootstrap the program state (see the example in Section 4.1, Figure 2 below).

In the simplest scenario, the user could theoretically *do nothing*, proceeding with the usual initialization, and just let the checkpoint data overwrite the default initial state. If this approach would carry a high overhead or be undesirable algorithmically, then development of one of the aforementioned initialization procedures is necessary.

Aside from failed tasks, any remaining tasks must also be reset to the checkpointed state to continue executing in cooperation with the restarted tasks. CUMULVS supports two distinct alternatives for this recovery of non-failed tasks: *restart* and *rollback*. If there is not significant overhead associated with restarting a new process and initializing its program state, then clearly the easiest option is to simply kill off and restart any non-failed tasks, precisely as would have been done if they had in fact failed. This approach does not require any additional special handling in the user application, beyond the checkpoint restart initialization described above.

If, however, it is cumbersome or costly to restart new tasks, or the initialization overhead (whether default or restart) is prohibitive, then it may be necessary to use "rollback" within the non-failed tasks. Rollback means that these tasks must reset their internal program state, using their existing local process space, to continue executing with any restarted tasks. This will likely require the development of a special re-initialization procedure to allow downloading of checkpoint data. This approach also requires substantial instrumentation of the user application to place "watch points" for restart notification wherever messages are received. The application tasks must check for restarts and be capable of unrolling the program call stack at any point to recover using a checkpoint. More details and issues of this procedure are described in the following example and in Section 4.3.

The next three subsections describe the user instrumentation process in more detail, including an example instrumentation, the actual library interface and some issues that the application programmer must face, respectively.

4.1 Example Instrumentation

To illustrate the usage of the CUMULVS user library for checkpointing, consider Figure 2 which shows a pseudo-code example. While CUMULVS provides both C and Fortran bindings for its user library, only the C bindings are shown here for brevity. In this example, the application initializes the standard CUMULVS visualization and steering system using the `stv_init()` call. It selects "`solver`" as its logical name for external viewer lookup, provides CUMULVS with message tag `100` for all of its internal communication[1], and indicates that the application has `ntasks` tasks, with this specific task being logical instance number `myinst`. The checkpointing `stvOptCpRecovery` option is set to `stvOptCpRollback`, so that application tasks will be rolled back rather than being killed and restarted in response to a failure. Checkpointing is then initialized using `stv_cpInit()`, which indicates that the executable file for restarting failed tasks in this application is "`parsolv`" and that message tag `101` should be used for messages to notify the application of restarts. The `&ntasks` reference is passed to allow CUMULVS to make adjustments if restarting from a checkpoint using a different number of tasks. The `restart` value returned also equals this `ntasks` value if the task is restarting from a checkpoint.

Application `solver` defines several parameters and a 3-dimensional data field `pressure`. These are all marked with `stvVisCp`, indicating that they are viewable/steerable and also selected as part of the program state for checkpointing. In a restart, `solver` must incrementally bootstrap itself by defining several fields and parameters for CUMULVS, filling these in with checkpoint data, and then using the values of these variables to define the remaining parameters for their checkpoint data update. Specifically, the size of the integer vector `ix` depends on the value for `del`, which is a checkpointed variable. Therefore, to correctly allocate the `ix` vector during restart from a checkpoint, a call to `stv_loadFromCp()` must first be made to set the restart value for `del`.

At the end of the main work loop, `solver` passes control to the CUMULVS `stv_sendToFE()` routine for visualization and steering handling. Every 10 iterations `stv_checkpoint()` is called to request that a checkpoint be saved. If a failure occurs while in `work()`, then restart from the last checkpoint is initiated using a call to `stv_loadFromCP()`. When the application's work is complete, a single call to `stv_cpFinished()` is made to disconnect from the CUMULVS checkpointing daemon. Without this call, the application could not exit normally without the CPD continuing to try and restart it from the last checkpoint!

4.2 Library Calls

This section details the calls to the various CUMULVS user library routines.

The `stv_init()` routine must be invoked before any of the other CUMULVS library routines:

```
int status = stv_init( char *app_name,
    int msgtag, int ntasks, int myinst )
```

[1]ignored if the underlying message-passing substrate supports context.

```
main()
{
        /* Initialize CUMULVS Vis & Steering System */
        stv_init( "solver", 100, ntasks, myinst );
        /* Set CUMULVS Recovery Option to Rollback */
        stv_setopt( stvDefault, stvOptCpRecovery,
                stvOptCpRollback );
        /* Initialize CUMULVS Checkpointing for this Task */
        /* - executable for restart is "parsolv" */
        /* - use message tag 101 for fault notification */
        /* - application has "ntasks" tasks total */
        restart = stv_cpInit( "parsolv", 101, &ntasks );
        . . .
        /* Define Decomposition for Main Data Array */
        did = stv_decompDefine( 3,
                { stvBlock, stvCyclic, stvCollapse },
                (global bounds), ... );
        /* Define Main Data Array "pressure" for CUMULVS */
        /* - flag for Vis & Checkpointing (stvVisCp) */
        fid = stv_fieldDefine( p1[ ][ ][ ], "pressure",
                did, (declared bounds), stvFloat,
                (index in decomp), stvVisCp );

        /* Define Scalar Parameter "delta" */
        /* - flag for Vis & Checkpointing (stvVisCp) */
        stv_paramDefine( "delta", &del, stvDouble, stvVisCp );
        /* Check Restart Status... (returned by stv_cpInit()) */
        if ( restart )
                /* Load Program Vars from Checkpoint Data */
                stv_loadFromCP();
        /* Allocate "indices" Vector, Using "del" as Size */
        /* ("del" could have been set from checkpoint data...) */
        ix = (int *) malloc( (100.0 / del) * sizeof(int) );
        /* Define "indices" Parameter for CUMULVS */
        stv_paramDefine( "indices", ix[ ], stvInt, stvVisCp );
        /* Load Newly-Defined Program Vars from Checkpoint */
        /* (like "ix" which depended on "del"...) */
        if ( restart )
                stv_loadFromCP();
        /* Otherwise, Default Initialization of Program Vars */
        else
                init_data();
        . . .
        /* Main Work Loop */
        do
        {
                /* Execute Computation for Current Time Step */
                cc = work( &timestep );
                /* If Failure Notification Received in work(), */
                /* Reset State by Loading it from Checkpoint */
                if ( !cc )
                        stv_loadFromCP();
                /* Otherwise, Proceed... */
                else {
                        /* Pass Control to CUMULVS, Vis & Steer */
                        /* ("new_params" = # of steer updates) */
                        new_params = stv_sendToFE();
                        /* Checkpoint Every 10 Time Steps */
                        if ( !( timestep % 10 ) )
                                stv_checkpoint();
                }
        }
        while ( !done );
        . . .
        /* Tell CUMULVS to Stop Checkpoint Recovery */
        /* So Task Can Exit Normally... */
        stv_cpFinished();
}
```

Figure 2: Example CUMULVS Instrumentation

This routine initializes the base CUMULVS visualization and steering system, and creates an entry for the application in a database for viewer lookup. The `app_name` argument provides a logical name for the application. When a user wishes to connect a viewer to the application, this is the name that will be used to look up and identify the desired application tasks. The `ntasks` argument tells CUMULVS how many tasks will initially be started for the application, so viewer programs will know when the application tasks have all registered with CUMULVS. The `myinst` argument indicates a logical task number for the given task in the application, as needed for internal bookkeeping.

There are several run-time options which can be set for CUMULVS using the `stv_setopt()` routine:

```
int oldvalue = stv_setopt( int what,
    int option, int value )
```

For checkpointing, the `stvOptCpRecovery` option (with `what` left as `stvDefault`) can be set to have the application tasks either rolled back (`stvOptCpRollback`) or killed off and restarted (`stvOptCpRestart`) in response to a fault or failure. These are the two primary checkpointing modes that the application programmer can select from when developing a fault-tolerant application. Setting `stvOptCpRollback` requires the user to implement on-the-fly handling of restart notification messages throughout the application. In response to each such restart message, the application must "unroll" its program stack, reload its state from a checkpoint, and then continue on using the same system process. By default, the simpler `stvOptCpRestart` mode is selected. When using this mode, CUMULVS will simply kill off and restart the entire application in response to any failure. This does not require significant modification of the application's program flow, aside from handling the alternate program startup initialization when restarting from a checkpoint.

At the start of each checkpointed application task there must be a call to `stv_cpInit()`:

```
int restart = stv_cpInit( char *aout_name,
    int notifytag, int *ntasks )
```

This call should always be made *after* the main CUMULVS initialization routine, `stv_init()`, as well as the selection of the checkpointing recovery mode using `stv_setopt()`. The `aout_name` argument is the name of the executable file that is to be used when restarting the task in the event of a failure. This need not be the same file as was used to originally run the task. Allowing a different executable for restart can potentially alleviate some of the "restart vs. default" startup initialization issues mentioned above. For a given application it might be easier to define two different versions of a task's executable - one for normal startup and one for checkpoint restart. The `notifytag` argument is used by applications that wish to be "rolled back" rather than killed off and restarted when a failure has occurred. In this mode, the local CPD will send an error notification message to any remaining tasks using the given `notifytag` message tag, to inform the tasks that they need to roll back and recover using a checkpoint. The `ntasks` argument to `stv_cpInit()` passes in the number of tasks to be coordinated for this application, and on restart will return the number of tasks that are actually being used for a given restart. The call to `stv_cpInit()` will start a CPD process on the local machine if one is not already running, and will set up communication between the task and the local CPD. The return value of `stv_cpInit()`, if greater than zero, indicates that the application is being restarted from a checkpoint. In this case the `restart` value returned is the number of application tasks involved in the particular restart incarnation, the same as is returned in `ntasks`.

The application tasks can invoke `stv_isCpRestart()` directly at any time after initializing checkpointing with CUMULVS (i.e., after the call to `stv_cpInit()`) to determine if the current instance of the task needs to restart from a checkpoint:

```
int restart = stv_isCpRestart()
```

As with `stv_cpInit()`, if restart is greater than zero indicating a restart in progress, then the `restart` return value is the number of tasks in the restarted application.

To define a contiguous data field in CUMULVS and mark it as part of the program state to be checkpointed, the `aflag` argument to the `stv_fieldDefine()` routine is used:

```
stv_fieldDefine( ..., int aflag )
```

Similarly, the corresponding aflag argument to `stv_particleFieldDefine()` or `stv_paramDefine()` can be set to include particle fields or parameters, respectively, in the checkpoint data. (The full specifications for these routines are described in [31].) To include a given data field or parameter for visualization and steering only, the value of `aflag` should be set to `stvVisOnly`. To checkpoint the value of the data field or parameter without making it available for visualization and steering, `aflag` should set to `stvCpOnly`. To allow visualization and steering, *and* to include the given variable in the collected checkpoint data, `aflag` should be set to `stvVisCp` (or equivalently "`stvVisOnly | stvCpOnly`").

At the point in the application task where the data fields are "consistent" (Section 1.1) and the user wants to save a checkpoint, a call is made to the `stv_checkpoint()` routine:

```
int info = stv_checkpoint()
```

The `info` value returned will equal `stvStatusOk` if the checkpoint submission proceeded correctly, meaning that the checkpoint data is en route to the local CPD. If some fault or failure occurred during the checkpoint, then `stvRestart` is returned to indicate that the task should roll back and restart from the last coherent checkpoint (or the task may simply wait to be killed at this point, if configured for `stvOptCpRestart`).

When an application task is restarting from a checkpoint, the loading of the checkpoint data into the desired program variables is done using the `stv_loadFromCP()` routine:

```
int nleft = stv_loadFromCP()
```

This routine requests the latest coherent checkpoint from the local CPD and then waits for the incoming checkpoint data to arrive and be loaded into user variables. Any data fields and parameters that have been defined for CUMULVS before this routine is invoked, such that CUMULVS knows where the proper variable storage resides, will be filled in using the available checkpoint data. The `nleft` value indicates the number of program variables and parameters that have yet to be updated using checkpoint data. If all downloaded checkpoint data has been inserted into user variables then `nleft` will be zero. Incremental CUMULVS data field and parameter declarations can be interspersed with checkpoint updates, if required for data fields that depend on the values of other parameters which are being updated by the checkpoint. For these cases, data field and parameter declarations (`stv_decompDefine()`, `stv_fieldDefine()` and `stv_paramDefine()`) can be made between repeated calls to `stv_loadFromCP()`, until all checkpoint data has been updated.

The last CUMULVS routine that each checkpointed application needs to invoke is `stv_cpFinished()`:

```
int restart = stv_cpFinished()
```

This routine informs the CPD that the given application task has completed its work and would like to exit normally (without being restarted, or triggering any other actions). The call to `stv_cpFinished()` is a *blocking* call that synchronizes all checkpointing tasks, to insure that all of these tasks finish checkpointing in unison. Without this synchronization, a variety of unrecoverable race conditions could occur.

4.3 Application Issues

The user must select between the two ways in which CUMULVS can respond to a failure, killing all tasks and performing a complete restart, or spawning only the required replacement tasks and signaling the remaining active tasks to roll back and reload from a checkpoint. The first method requires no significant changes to the program flow, aside from the inevitable initialization handling to either start up normally or restart from a checkpoint. The second method requires the programmer to check at *every* message receipt for a restart notification from the local CPD, and on restart to manually unroll the current program stack. Each message must be checked for restarts because the failure of *any* cooperating task might preempt an expected message, leaving the given task potentially blocked.

If the overhead to restart an application task is prohibitive, or the initialization of non-checkpointed data fields requires significant computation by the application tasks, then the rollback option may be a necessity. In this case, no matter where in the subroutine hierarchy the given application task is executing, it must stop and return the failure notification up through the call stack. This requires checking the restart status not only after every receive call, but after each call to a subroutine that posts receives. At the top level, the application must then clean out any pending message queues and reset any open file pointers, so that execution can correctly continue from the point of the last checkpoint. Any messages left over from before the recovery could lead to erroneous behavior, and similarly with any hidden state implicit in open file pointers. For user applications written in C++, this ordeal can be handled substantially easier using trapping to unroll the program stack (see [27, 28]). In any case, CUMULVS can only handle the details of loading the checkpoint data into the application's variable storage - the rest is up to the user to coordinate.

An additional issue regarding the rollback scenario involves the identification of fault notification messages versus regular user messages. All notify messages will be tagged from the CPD using the `notifytag` as provided by the user application in its call to `stv_cpInit()`. Yet it is not necessarily a trivial matter to post a single blocking receive for *two* different message tags at once... The task cannot simply check for fault notifies before blocking on the regular expected messages, because a failure could occur between those actions. Continuously polling between the two different message tags is inefficient and not clean coding style. MPI supports a mechanism for handling this problem, using `MPI_Irecv()` and `MPI_Waitany()` to asynchronously check for multiple incoming messages. Note, however, that such a mechanism *must* be utilized *everywhere* that a task would block waiting to receive a message.

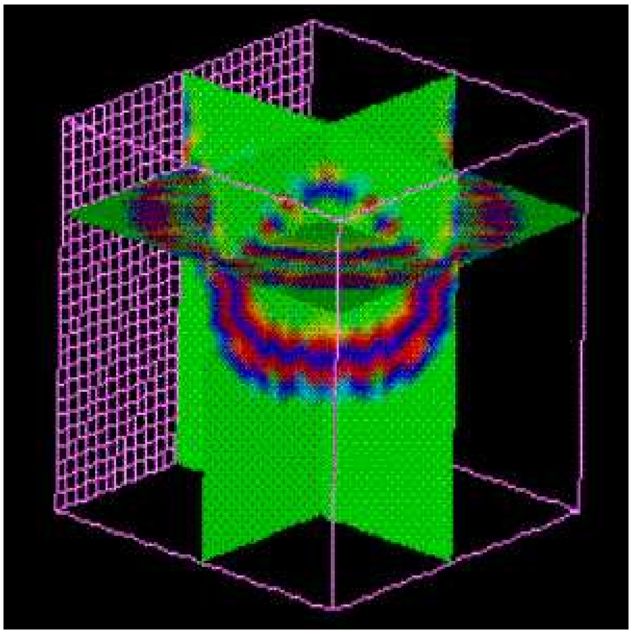

Figure 3: Finite Difference Example - Seismic Simulation

4.4 Case Study

CUMULVS checkpointing has been applied to instrument two production scientific applications, for fault tolerance and heterogeneous migration. The first application generates a synthetic seismic dataset by simulating the propagation of an acoustic signal through a layered media, using a finite difference approximation (see Figure 3). The second is a computational fluid dynamics (CFD) simulation that computes the pressure of the air flowing around a wing (see Figure 4). Both applications are written in Fortran using PVM as an underlying message-passing substrate. To determine the usefulness of the CUMULVS approach, several analyses were performed using these two applications.

First, the number of additional source code statements required to instrument the applications for CUMULVS was examined, including declarations of program variable semantics, initialization code for restarting from a checkpoint, and program flow changes (for both restart and rollback). All necessary statements were counted, including CUMULVS library calls and their surrounding code (such as `if`, `else`, `endif`). The results are presented in Table 5.

For the seismic simulation, there were a total of 51 statements to initialize the base CUMULVS visualization and steering system and instrument the main data field and 4 steering parameters. This counts all header file includes, temporary variable declarations, the call to `stv_init()`, and the decomposition, data field and parameter descriptions. (The code necessary to handle steering parameter updates, as utilized to introduce seismic "thumps" into the simulation, required 37 extra statements not counted here.) To add basic fault-tolerant restart capabilities and collect checkpoints every `N` iterations, an additional 21 statements were required. These statements described a second data array, 7 data vectors, and 5 additional scalar parameters to be checkpointed, and modified the default program initialization to handle restart from a checkpoint. To implement "rollback-capable" restarts, an additional 41 statements were added or modified to catch notify messages and unroll the program

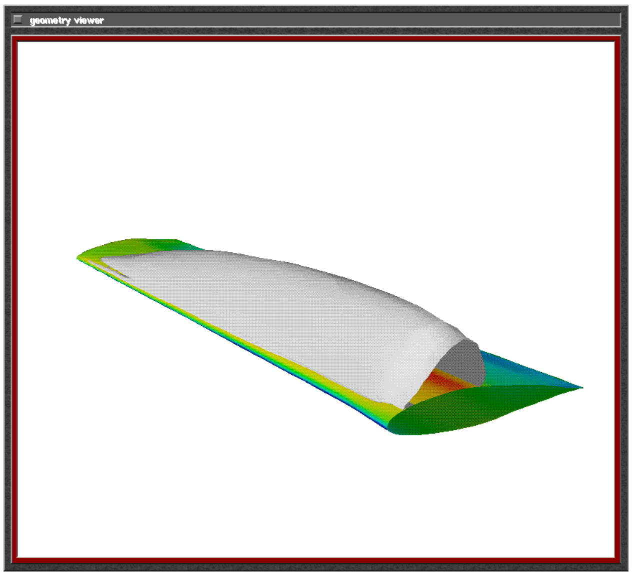

Figure 4: CFD Example - Air Flow Around a Wing

Seismic:	
Vis / Steer System Init	3
Vis / Steer Var Decls	48
CP Restart Initialization	21
CP Rollback Handling	41
Flow Around Wing:	
Vis / Steer System Init	3
Vis / Steer Var Decls	73
CP Restart Initialization	12
CP Rollback Handling	34

Figure 5: Additional Source Code Statements

Experiment	SGI	Cluster	Hetero
Seismic - No Checkpointing	2.83	6.23	9.46
Seismic - Checkpoint Restart	2.99	6.50	10.76
Seismic - Checkpoint Rollback	3.03	6.66	10.90
Wing - No Checkpointing	0.69	1.58	6.14
Wing - Checkpoint Restart	0.77	1.71	7.10
Wing - Checkpoint Rollback	0.79	1.71	7.30

Figure 6: Checkpointing's Impact on Performance

stack. Overall, the full visualization, steering, and fault-tolerant rollback instrumentation accounted for less than 1% of the resulting code - only 204 extra lines of code out of 20,836.

For the simulation of flow over the wing, there were a total of 76 statements to initialize the base CUMULVS visualization and steering system and instrument 13 data fields, with 2 decomposition types, and 6 steering parameters. (The code necessary to handle steering parameter updates for this application amounted to 28 extra statements not counted here.) To add basic fault-tolerant restart capabilities and collect checkpoints every N iterations, only an additional 12 statements were required. These statements described one additional scalar parameter to be checkpointed, in addition to the data fields and parameters previously defined for visualization that were also to be checkpointed. The 12 additional checkpointing statements included all necessary modifications to the default program initialization for handling restart recovery from a checkpoint. To implement rollback recovery, an additional 34 statements were added or modified around messaging routines, and to unroll the program stack. Overall, the full CUMULVS instrumentation for this application totaled a bit more than the seismic, at just under 8% of the total instrumented code - 188 extra lines of code out of 2438. However, the checkpointing portion of the instrumentation for this application was significantly smaller, primarily due to the extensive visualization and steering instrumentation that had already been applied.

The other analysis for this case study involved the collection of some basic application execution timings, to evaluate the efficiency of the CUMULVS checkpoint collection and recovery system. The experiments were run using several different sets of resources, specifically on 4 nodes of an 8 node 233 MHz dual-Pentium Linux cluster, a 4-R10000 node SGI Onyx2 multiprocessor, and on a heterogeneous collection of Sun Sparc5, IBM RS6000, SGI R10000 Octane and Pentium Linux machines. Timings were collected to show application performance without any CUMULVS instrumentation (baseline), and then with checkpoints collected every 20 iterations, using both restart and rollback recovery modes. The impact to application performance can be seen in Table 6 which shows seconds per application iteration, as averaged over 100 iterations.

For the Seismic simulation, adding simple restart checkpointing recovery incurred from 4% to 14% overhead compared to the baseline without any checkpointing. Further instrumentation of the application messaging for rollback recovery added an extra 1% to 3% overhead. The size of a full checkpoint for this application was approximately 14 Megabytes. For the Wing Flow simulation, the basic restart checkpointing added between 8% and 15% overhead, and applying rollback recovery added from 0% to 2.5% overhead. The checkpoints for this application were only about 6 Megabytes each.

Both of these simulations use communication only to exchange data at the end of iterative computational phases, with the Seismic code exchanging data twice per iteration using 100 Kilobyte messages, and the Flow code exchanging data once per iteration using smaller 13 Kilobyte messages. This accounts for the minor additional overhead experienced when the application's communication is instrumented for rollback recovery. More communication-bound applications would expect a more significant slowdown for this rollback instrumentation.

In general, for homogeneous systems with faster communication hardware, the basic restart checkpointing impacts performance by only 5% to 8%. On heterogeneous collections of machines, with varying levels of CPU performance and only standard Ethernet connectivity, the degradation due to checkpointing increases slightly. This is likely due to network loading that delays the application tasks in sending out their checkpoint data, and could be an effect of processor load imbalance. It should be noted that, even on homogeneous systems, CUMULVS checkpoints are collected using a default data encoding to allow heterogeneous migration and restart, so no additional performance is directly gained from homogeneity.

5 Future Work and Status

In terms of checkpointing capabilities, CUMULVS is really still in its infancy. There are many extensions that could be made to the prototype system to improve its usability, automation and performance. For example, many of the elaborate techniques and mechanisms applied to optimize traditional checkpointing systems [32] could be applied to the CUMULVS model. Improved user interfaces or GUIs, as well as integrated development environments, could be used to expedite the user specification of program state variables and consistent checkpoint collection. Also, additional assistance could be provided for manipulating various external state, including files.

CUMULVS presently supports only serial "viewer" programs that connect to potentially parallel application programs. And although the CUMULVS CPD tasks make up a parallel application, currently user processes contact only their local CPD task. A powerful generalization would be to allow parallel programs to connect to and interact with other parallel programs. The CUMULVS-style connection protocols, with the underlying library handling all of the details, would allow development of parallel-to-parallel agents for steering, visualization, checkpointing or other types of coupling or interaction. It will take significant analysis to keep these connection protocols reliable and recoverable in the parallel-to-parallel case. However, the resulting interconnectivity could open the door to a large number of new coupled applications.

One important requirement for this capability is a set of efficient routines to transform and redistribute decomposed data. For example, a simulation program may store a data field in a Block-Cyclic distribution across 16 processors, while a parallel visualization program may desire some subset of this data in a 4 processor Block distribution. The existing one-to-one and parallel-to-serial transformations for collecting distributed data must be extended to support new parallel-to-parallel transformations.

CUMULVS version 1.1 supports the preliminary checkpointing interface. It is now available, including source and user's guide, via the CUMULVS home page at "http://www.epm.ornl.gov/cs/cumulvs.html". Online support is available by sending email to "cumulvs@msr.epm.ornl.gov". Future releases may include improved user interface tools to expedite the CUMULVS instrumentation process for new and existing user applications.

References

[1] G. A. Geist, J. A. Kohl, P. M. Papadopoulos, "CUMULVS: Providing Fault-Tolerance, Visualization and Steering of Parallel Applications," *International Journal of Supercomputing Applications*, also available via `http://www.epm.ornl.gov/cs/cumulvs96.ps`.

[2] P. M. Papadopoulos, J. A. Kohl, "CUMULVS: an Infrastructure for Steering, Visualization and Checkpointing for Parallel Applications," 1996 PVM User's Group Meeting, Santa Fe, NM.

[3] J. A. Kohl, P. M. Papadopoulos, "The Design of CUMULVS: Philosophy and Implementation," 1996 PVM User's Group Meeting, Santa Fe, NM.

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

[5] J. A. Clarke, J. J. Hare, C. E. Schmitt, "Distributed Interactive Computing Environment (DICE)," Army Research Laboratory, Major Shared Resource Center, see `http://frontier.arl.mil/clarke/dice.html`.

[6] J. A. Clarke, J. J. Hare, C. E. Schmitt, "Dice Data Directory (DDD)," Army Research Laboratory, Major Shared Resource Center, see `http://frontier.arl.mil/clarke/Dd.html`.

[7] "Hierarchical Data Format (HDF)," National Center for Supercomputing Applications, see `http://hdf.ncsa.uiuc.edu/`.

[8] 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*, MIT Press, Cambridge, MA, 1994.

[9] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. *MPI: The Complete Reference*, MIT Press, Cambridge, MA, 1996.

[10] R. Armstrong, P. Wyckoff, C. Yam, M. Bui-Pham, N. Brown, "Frame-Based Components for Generalized Particle Methods," High Performance Distributed Computing (HPDC '97), Portland, OR, August 1997 (formerly "POET," see `http://glass-slipper.ca.sandia.gov/ rob/poet/`).

[11] J. S. Plank, "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance," Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, Knoxville, TN, July 1997.

[12] K. M. Chandy, L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," *ACM Transactions on Computer Systems*, Vol. 3, No. 1, February 1985, pp. 63-75.

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

[14] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, J. Walpole, "MPVM: A Migration Transparent Version of PVM," *Computing Systems*, Vol. 8, No. 2, Spring 1995, pp. 171-216.

[15] J. Casas, D. Clark, P. Galbiati, R. Konuru, S. Otto, R. Prouty, J. Walpole, "MIST: PVM with Transparent Migration and Checkpointing," 1995 PVM Users' Group Meeting, Pittsburgh, PA.

[16] Y. Chen, J. Plank, K. Li, "CLIP: A Checkpointing Tool for Message-Passing Parallel Programs," SC97: High Performance Computing & Networking, San Jose, CA, November 1997.

[17] J. Leon, A. Fisher, P. Steenkiste, "Fail-Safe PVM: A Portable Package for Distributed Programming with Transparent Recovery," Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, February 1993.

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

[19] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System," *Communications of the ACM*, Vol. 39, No. 4, April 1996, pp. 54-63.

[20] M. Litzkow, T. Tannenbaum, J. Basney, M. Livny, "Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System," Computer Sciences Technical Report #1346, University of Wisconsin-Madison, April 1997.

[21] K. Li, J. Naughton, J. Plank, "Low-Latency, Concurrent Checkpointing for Parallel Programs," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 8, August 1994, pp. 874-879.

[22] A. Beguelin, E. Seligman, P. Stephan, "Application Level Fault Tolerance in Heterogeneous Networks of Workstations," *Journal of Parallel and Distributed Computing*, Special Issue on Workstation Clusters and Network-based Computing, June 1997.

[23] L. Silva, J. Silva, S. Chapple, L. Clarke, "Portable Checkpointing and Recovery," Proceedings of the Fourth IEEE International Symposium on High-Performance Distributed Computing (HPDC '95), Washington, D.C., August 1995.

[24] A. Baratloo, P. Dasgupta, Z. Kedem, "Calypso: A Novel Software System for Fault-Tolerant Parallel Processing on Distributed Platforms," Proceedings of the Fourth IEEE International Symposium on High-Performance Distributed Computing (HPDC '95), Washington, D.C., August 1995.

[25] Y. Huang, C. Kintala, Y-M. Wang, "Software Tools and Libraries for Fault Tolerance," IEEE Technical Committee on Operating Systems and Application Environments, Vol. 7, No. 4, Winter 1995, pp. 5-9.

[26] D. Cummings, L. Alkalaj, "Checkpoint / Rollback in a Distributed System Using Coarse-Grained Dataflow," Proceedings of the 24th International Symposium on Fault-Tolerant Computing, Austin, TX, June 1994, pp. 424-433.

[27] A. J. Ferrari, S. J. Chapin, A. S. Grimshaw, "Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification," Technical Report CS-97-05, Department of Computer Science, University of Virginia, Charlottesville, VA, March 25, 1997.

[28] A. J. Ferrari, "Process Introspection: A Checkpoint Mechanism for High Performance Heterogeneous Distributed Systems," Technical Report CS-96-15, Department of Computer Science, University of Virginia, Charlottesville, VA, October 10, 1996.

[29] J. A. Kohl, P. M. Papadopoulos, "Fault-Tolerance and Reconfigurability Using CUMULVS," Cluster Computing Conference, Emory University, Atlanta, GA, March 9-11, 1997.

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

[31] J. A. Kohl, P. M. Papadopoulos, "CUMULVS User's Guide: Computational Steering and Interactive Visualization in Distributed Applications," Technical Report ORNL/TM-13299, Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, August 1996.

[32] J. Plank, M. Beck, G. Kingsley, K. Li, "Libckpt: Transparent Checkpointing under Unix," Conference Proceedings, Usenix Winter 1995 Technical Conference, New Orleans, LA, January 1995.