Implementation and Evaluation of a Scalable Application-level Checkpoint-Recovery Scheme for MPI Programs

Martin Schulz, Greg Bronevetsky, Rohit Fernandes,
Daniel Marques, Keshav Pingali, Paul Stodghill
Department of Computer Science,
Cornell University, Ithaca, NY 14853

ABSTRACT

The running times of many computational science applications are much longer than the mean-time-to-failure of current high-performance computing platforms. Therefore, to run to completion, these applications must tolerate hardware failures.

Checkpoint-and-restart (CPR) is the most commonly used scheme for accomplishing this - the state of computation is saved periodically on stable storage, and when a hardware failure is detected, the computation is restarted from the most recently saved state. Most automatic CPR schemes in the literature can be classified as blocking, system-level checkpointing schemes because they take core-dump style snapshots of the computational state when all the processes are blocked at global barriers in the program. Unfortunately, a system that implements this style of checkpointing is tied to a particular platform; in addition, it cannot be used if there are no global barriers in the program.

In our research project, we are exploring an alternative called non-blocking application-level checkpointing. In our approach, programs are transformed by a pre-processor so that they become self-checkpointing and self-restartable on any platform; there is also no assumption about the existence of global barriers in the code.

In this paper, we describe our implementation of non-blocking application-level checkpointing. We present experimental results on both a Windows cluster and the Lemieux system at the Pittsburgh Supercomputer Center, and argue that these results demonstrate both the platform-independence and the scalability of our approach.

1. INTRODUCTION

The high-performance computing community has largely ignored the problem of implementing software systems that can tolerate hardware failures. This is because until recently, most parallel computing was done on relatively reliable big-iron machines whose mean-time-between-failures (MTBF) was much longer than the execution time of most programs. However, the burgeoning size and complexity of parallel machines is increasing the probability of hardware failures, making it imperative that parallel programs tolerate such failures.

The most popular technique for accomplishing this is called checkpoint-and-restart (CPR for short). Most programmers implement CPR manually by (i) identifying points in the program where the amount of state that needs to be saved is small, (ii) determining what data must be saved at each such point, and (iii) inserting code to save that data on disk and restart the computation after failure. For example, in a protein-folding code using *ab initio* methods, programmers save the positions and velocities of the bases (and a few variables such as the time step number) at the end of each time step. To ensure a consistent view of global data structures, this approach of *manual application-level checkpointing* requires global barriers at the points where state is saved. Although barriers are present in parallel programs that are written in a bulk-synchronous manner [11], many other programs such as the HPL benchmark [15] and some of the NAS Parallel Benchmarks do not have global barriers.

A different approach to CPR, developed by the distributed systems community, is *system-level* checkpointing, in which all the bits of the computation are periodically saved on stable storage. This is the approach used in the Condor system [14] for taking uniprocessor checkpoints, for example. The amount of saved state can be reduced by using incremental state saving, but it is usually many orders of magnitude more than is saved by manual application-level checkpointing. For parallel programs, the problem of taking a system-level checkpoint reduces to the uniprocessor problem if there are global barriers where state can be saved and there are no messages in flight across these barriers. Without global synchronization, it is not obvious when the state of each process should be saved so as to obtain a global snapshot of the parallel computation. One possibility is to use coordination protocols such as the Chandy-Lamport [7] protocol.

The relative advantages and disadvantages of manual application-level checkpointing (ALC) and automatic system-level checkpointing (SLC) can be summarized as follows. Manual ALC requires programmer intervention, but it has the advantage that the resulting programs are self-checkpointing and self-restartable on any platform, and usually save only a small amount of data to disk. SLC is automatic, but systems that use SLC are platform-specific and usually save a lot more data at a checkpoint than programs that use ALC.

In principle, one could get the best of both worlds with *automatic* application-level checkpointing. This requires a pre-compiler that

[0] This work was supported by NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-0121401.

SC'04, November 6—12, 2004, Pittsburgh, Pennsylvania, USA.

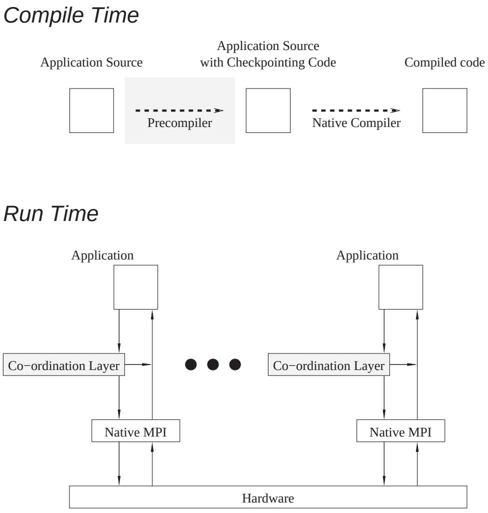

Figure 1: System Architecture

can automatically transform a parallel program into one that can checkpoint its own state and restart itself from such a checkpoint. To handle programs without global barriers, we need a protocol for coordinating checkpointing by different processes.

In this paper, we describe the implementation of such a system, and evaluate its scalability on large multiprocessor platforms.

Figure 1 is an overview of our approach. The C^3 (Cornell Checkpoint (pre-)Compiler) reads almost unmodified C/MPI source files and instruments them to perform application-level state-saving; the only additional requirement for programmers is that they must insert a `#pragma ccc_checkpoint` at points in the application where checkpoints might be taken. At runtime, some of these pragmas will force checkpoints to be taken at that point, while others will take a checkpoint there only if a timer has expired or if some other process has initiated a global checkpoint. The output of this precompiler is compiled with the native compiler on the hardware platform, and is linked with a library that constitutes a *co-ordination layer* for implementing the non-blocking coordination. This layer sits between the application and the MPI library, and intercepts all calls from the instrumented application program to the MPI library. Note that MPI can bypass the co-ordination layer to read and write message buffers in the application space directly. Such manipulations, however, are not invisible to the protocol layer. MPI may not begin to access a message buffer until after it has been given specific permission to do so by the application (e.g. via a call to `MPI_Irecv`). Similarly, once the application has granted such permission to MPI, it should not access that buffer until MPI has informed it that doing so is safe (e.g. with the return of a call to `MPI_Wait`). The calls to, and returns from, those functions are intercepted by the protocol layer.

This design permits us to implement the coordination protocol without modifying the underlying MPI library, which promotes modularity and eliminates the need for access to MPI library code, which is proprietary on some systems. Furthermore, the instrumented programs are self-checkpointing and self-restarting on any platform. The entire runtime system is written entirely in C and use only a very small set of system calls. Therefore, the C^3 system is portable among different architectures and OSs; currently, it has been tested on x86 and PPC Linux, Sparc Solaris, x86 Win32, and Alpha Tru64. This is in contrast to a typical system-level CPR system, which needs to deal with the specifics of a machine's register file, stack layout, argument passing convention, etc.

The rest of this paper is organized as follows: In Section 2, we will enumerate some of the problem involved in providing ALC for MPI applications. In Section 3, we will present our basic protocol for non-blocking, coordinating ALC. This protocol is based on results in our earlier papers [4, 5], but it incorporates several significant improvements and extensions that were developed during our first complete implementation. In particular, the approach to coordinating checkpoints is completely different. In Section 4, we will expand the basic protocol to cover advanced features of MPI. In Section 5, we describe how each process saves its computational state. In Section 6, we present experimental performance results for the C^3 on several systems and several applications. In Section 7, we compare our work with related work in the literature. In Section 8, we present our conclusions and discuss our future work.

2. DIFFICULTIES IN APPLICATION-LEVEL CHECKPOINTING OF MPI PROGRAMS

In this section, we describe the difficulties with implementing application-level, coordinated, non-blocking checkpointing for MPI programs. In particular, we argue that existing protocols for non-blocking parallel checkpointing, which were designed for system-level checkpointers, are not suitable when the state saving occurs at the application level. In Section 3, we show how these difficulties are overcome with our approach.

2.1 Terminology

In our system, a global checkpoint can be initiated by any process in the program. To participate in taking the global checkpoint, every other process saves its local computational state, and some book-keeping information on stable storage. The collection of local computational states and book-keeping information is called a *recovery line*.

In our approach, recovery lines do not cross each other. The execution of the program can therefore be divided into a succession of *epochs* where an epoch is the interval between two successive recovery lines (by convention, the start of the program is assumed to begin the first epoch). Epochs are labeled successively by integers starting at zero, as shown in Figure 2.

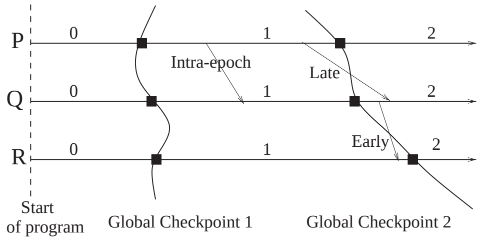

Figure 2: Epochs and message classification

It is convenient to classify an application message into three categories depending on the epoch numbers of the sending and receiving processes at the points in the application program execution when the message is sent and received respectively.

DEFINITION 1. *Given an application message from process A to process B, let e_A be the epoch number of A at the point in the application program execution when the send command is executed, and let e_B be the epoch number of B at the point when the message is delivered to the application.*

- Late message: *If $e_A < e_B$, the message is said to be a late message.*
- Intra-epoch message: *If $e_A = e_B$, the message is said to be an intra-epoch message.*
- Early message: *If $e_A > e_B$, the message is said to be an early message.*

Figure 2 uses the execution trace of three processes named P, Q and R to show examples of these three kinds of messages. The source of the arrow represents the point in the execution of the sending process at which control returns from the MPI routine that was invoked to send this message. Similarly, the destination of the arrow represents the delivery of the message to the application program. An important property of the protocol described in Section 3 is that an application message can cross at most one recovery line. Therefore, in our system, e_A and e_B in Definition 1 can differ by one at most.

In the literature, late messages are sometimes called *in-flight* messages, and early messages are sometime called *inconsistent* messages. This terminology was developed in the context of system-level checkpointing protocols; in our opinion, it is misleading in the context of application-level checkpointing.

2.2 Delayed State-saving

A fundamental difference between system-level checkpointing and application-level checkpointing is that a system-level checkpoint may be taken at any time during a program's execution, while an application-level checkpoint can only be taken when program execution encounters a `ccc_checkpoint` pragma.

System-level checkpointing protocols, such as the Chandy-Lamport distributed snapshot protocol, exploit this flexibility with checkpoint scheduling to avoid the creation of early messages—during the creation of a global checkpoint, a process P must take its local checkpoint before it can read a message from process Q that was sent after Q took its own checkpoint. This strategy does not work for application-level checkpointing, because process P might need to receive an early message before it can arrive at a point where it may take a checkpoint.

Therefore, unlike system-level checkpointing protocols which typically handle only late messages, application-level checkpointing protocols must handle both late and early messages.

2.3 Handling Late and Early Messages

We use Figure 2 to illustrate how late and early messages must be handled.

Suppose that one of the processes in this figure fails after Global Checkpoint 2 is taken. For process Q to recover correctly, it must obtain the late message that was sent to it by process P prior to the failure. Therefore, we need mechanisms for (i) identifying late messages and saving them along with the global checkpoint, and (ii) replaying these messages to the receiving process during recovery. In our implementation, each process uses a `Late-Message-Registry` to save late messages. Each entry in this registry contains the message *signature*,

$$< sending\ node\ number, tag, communicator >$$

and the message data. There might be multiple messages with the same signature in the registry, and these are maintained in the order in which they are received. Once recording is complete, the contents of this `Late-Message-Registry` are saved on stable storage.

Early messages, such as the message sent from process Q to process R pose a different problem. On recovery, process R does not expect to be resent this message, so process Q must suppress sending it. To handle this, we need mechanisms for (i) identifying early messages, and (ii) ensuring that they are not resent during recovery. In our implementation, each process uses a `Early-Message-Registry` to record the signatures of early messages. Once all early messages are received by a process, the `Early-Message-Registry` is saved on stable storage. During recovery, each process sends relevant portions of its `Early-Message-Registry` to other processes to tell them which messages need to be suppressed. Each process constructs a `Was-Early-Registry` from the information it receives from all other processes, and suppresses the matching message sends during recovery.

Early messages pose another more subtle problem. In Figure 2, the saved state of process R at Global Checkpoint 2 may depend on data contained in the early message from process Q. If the contents of that message depend on the result of a non-deterministic event at Q, such as a wild-card receive, that occurred after Q took its checkpoint, that event must be re-generated in the same way during recovery.

Therefore, mechanisms are needed to (i) record the non-deterministic events that a global checkpoint depends on, so that (ii) these events can be replayed during recovery.

2.4 Problems Specific to MPI

In addition to the problems discussed above, problems specific to MPI must be addressed.

Many of the protocols in the literature such as the Chandy-Lamport protocol assume that communication between processes is FIFO. In MPI, if a process P sends messages with different tags or communicators to a process Q, Q may receive them in an order different from the order in which they were sent. It is important to note that this problem has nothing to do with FIFO behavior or lack thereof in the underlying communication system; rather, it is a property of the order in which an application chooses to receive its messages.

MPI also supports a very rich set of group communication calls called collective communication calls. These calls are used to do broadcasts, perform reductions, etc. In MPI, processes do not need to synchronize to participate in any collective communication call other than barrier. Therefore, the problem with collective calls is that in a single collective call, some processes may invoke the call before taking their checkpoints while other processes may invoke the call after taking their checkpoints. Unless something is done, only a subset of the processes will re-invoke the collective call during recovery, which would be incorrect.

Finally, the MPI library has internal state that needs to be saved as part of an application's checkpoint. For example, when a process posts a non-blocking receive, the MPI library must remember the starting address of the buffer where the data must be written when it arrives, the length of the buffer, etc. If a process takes a checkpoint in between the time it posts a non-blocking receive and when the message is actually received by the application layer, the checkpoint must contain relevant information about the pending non-blocking receive so that the message can be received correctly after recovery. Previous work has investigated modifying the MPI library code [18], or providing a specifically designed implementation of the library [1], but these strategies are not portable.

3. A NON-BLOCKING, COORDINATED PROTOCOL FOR APPLICATION-LEVEL CHECKPOINTING

We now describe the coordination protocol we use for coordi-

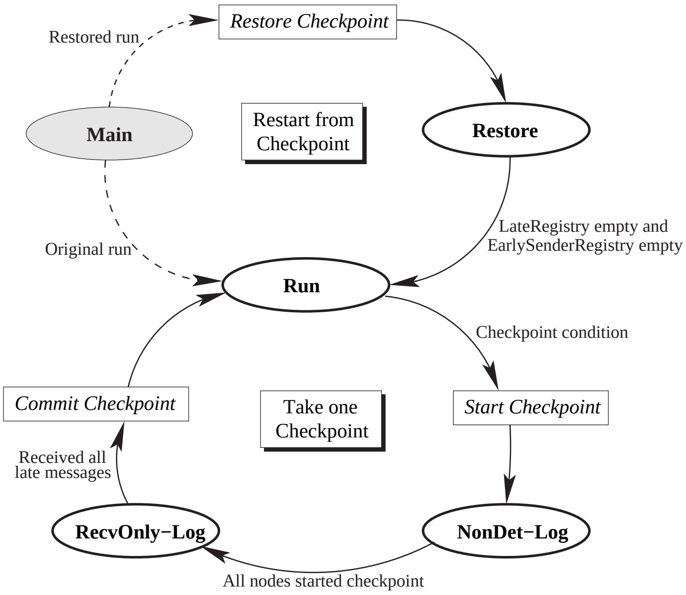

Figure 3: State transitions of a process

nated, application-level checkpointing. The protocol is independent of the technique used by processes to save their computational state. To avoid complicating the presentation, we first describe the protocol for blocking point-to-point communication only. In Section 4, we describe how advanced features of MPI such as asynchronous communication, arbitrary datatypes, and collective communication can be handled by using these mechanisms.

3.1 High-level Description of Protocol

At any point during execution, a process is in one of the states shown in the state transition diagram of Figure 3. We call these states *modes* in our description. Each process maintains variables named `Epoch` and `Mode` to keep track of its current epoch and mode.

In addition, each process maintains a number of variables to capture the state of the communication operations to determine whether it has received all outstanding messages from other processes. Each process maintains an array `Send-Count` containing one integer for each process in the application; `Sent-Count`[Q] is the number of messages sent from the local process to Q in the current epoch. Furthermore, a process maintains a set of counters to capture the number and type (late, intra-epoch, early) of all received messages. These variables are updated at every communication operation as shown in Figure 4, and are explained later.

The square boxes in Figure 3 show the actions that are executed by a process when it makes a state transition; pseudo-code for these actions is shown in Figure 5. These states and transitions are described in more detail next.

Run During normal execution, a process is in the **Run** mode. As described above, it increments `Sent-Count`[Q] when it sends a message to process Q; when it receives an early message, it adds it to its `Early-Message-Registry`.

A process takes a checkpoint when it reaches a pragma that forces a checkpoint. Alternatively, it may get a control message called *Checkpoint-Initiated* from another process which has started its own checkpointing; in this case, the process continues execution until it reaches the next pragma in its own code, and then starts its checkpoint. These conditions are described in the code for the pragma shown in Figure 5.

To take a checkpoint, the function called `chkpt_StartCheckpoint` in Figure 5 is invoked. For now, it is sufficient to note that this function saves the computational state of the process, and its `Early-Message-Registry` on stable storage, and sends a *Checkpoint-Initiated* message to every other process Q, sending the value of `Sent-Count[Q]` with this message. It then re-initializes the `Sent-Count` array and the `Early-Message-Registry`, and transitions to the **NonDet-Log** state, beginning a new epoch.

NonDet-Log In this mode, the process updates counters and registries as in the **Run** mode, but it also saves non-deterministic events and late messages in the `Late-Message-Registry` for replay during recovery.

In addition, the signature (but not the data) of each intra-epoch message is saved in the `Late-Message-Registry`. This enables the non-determinism of wild-card receives (`MPI_ANY_SOURCE` and/or `MPI_ANY_TAG`) to be replayed correctly on recovery.

When the process gets a *Checkpoint-Initiated* message from *all* other processes, it knows that every process has started a new epoch, so any message it sends from that point on will not be an early message. Therefore, it terminates the logging of non-deterministic events and transitions to the `RecvOnly-Log` mode. It must also perform this transition if it receives a message from a process Q that has itself stopped logging non-deterministic events; intuitively, this is because it knows that Q knows that all processes have taken their local checkpoints.

The second condition for performing the transition is a little subtle. Because we make no assumptions about message delivery order, it is possible for the following sequence of events to happen. Process P stops logging non-deterministic events, makes a non-deterministic decision, and then sends a message to process Q containing the result of making this decision. Process Q could use the information in this message to create another non-deterministic event; if Q is still logging non-deterministic events, it stores this event, and hence, the saved state of the global computation is causally dependent on an event that was not itself saved. To avoid this problem, we require a process to stop logging non-deterministic events if it receives a message from a process that has itself stopped logging non-deterministic events.

RecvOnly-Log In this state, the process continues to log late messages in the `Late-Message-Registry`.

When the process receives all late messages from the previous epoch, it invokes `chkpt_CommitCheckpoint`, which is shown in Figure 5. This function writes the `Late-Message-Registry` to stable storage. The process then transitions to the **Run** state.

Restore A process recovering from failure starts in the `Restore` state, and invokes `chkpt_RestoreCheckpoint`, which is shown in Figure 5. It sends every other process Q the signatures of all early messages that Q sent it before failure, so that these sends can be suppressed during recovery. Each process collects these signatures into a `Was-Early-Registry`. During recovery, any message send that matches

```
chkpt_MPI_Send()
    If (Mode=NonDetLog)
        Check for control messages
        If (all nodes have started checkpoints)
            Mode:=RecvOnly-Log

    If (Mode!=Restore)
        PiggyBackData:=(Mode,Epoch)
        MPI_Send(<original send parameters>)
        Sent-Count[Target]++

    Else /* Mode must be Restore */
        If (parameters match entry in Was-Early-Registry)
            Remove entry from Was-Early-Registry
            If  (Late-Message-Registry is empty) and
                (Was-Early-Registry is empty)
                    Mode:=Run
            return MPI_SUCCESS
        Else
            PiggyBackData:=(Mode,Epoch)
            MPI_Send(<original send parameters>)
            Sent-Count[Target]++

chkpt_MPI_Recv()
    If (Mode!=Restore)
        If (Mode=NonDetLog)
            Check for control messages
            If (all nodes have started checkpoints)
                Mode:=RecvOnly-Log

        MPI_Recv((<original recv parameters>)

        If (MsgType=Early)
            Early-Received-Counter[Source]++
            Add to Early-Message-Registry
        If (MsgType=Intra-epoch)
            Received-Counter[Source]++
            If (Sender is Logging) and (mode=NonDetLog)
                Add signature to Late-Message-Registry
        If (MsgType=Late)
            Late-Received-Counter[Source]++
            Add message to Late-Message-Registry

    Else /* Mode must be Restore */
        If (parameters match a message in Late-Message-Registry)
            Restore message from disk
            Delete entry in Late-Message-Registry
            If  (Late-Message-Registry is empty) and
                (Was-Early-Registry is empty)
                    Mode:=Run
        Else
            If (parameters match an entry in Late-Message-Registry)
                Restrict parameters to those in the registry
                Delete entry in Late-Message-Registry
                If  (Late-Message-Registry is empty) and
                    (Was-Early-Registry is empty)
                        Mode:=Run
                MPI_Recv(<modified recv parameters>)
            Else
                MPI_Recv(<original recv parameters>)
            Received-Counter[Source]++
            If (Message is from next epoch,i.e., is early)
                Early-Received-Counter[Source]++
                Add Message to Early-Message-Registry
            Else
                Received-Counter[Source]++
```

Figure 4: Wrapping Communication calls

```
#pragma ccc_checkpoint
    If (mode=Run)
        Check for control messages
        If  (checkpoint forced) or
            (timer expired) or
            (at least one other node has started a checkpoint)
                Call chkpt_StartCheckpoint
                If (all nodes have started checkpoint)
                    If (no late messages expected)
                        Mode:=Run
                    Else
                        Mode:=Recv-Log
                Else
                    Mode:=NonDet-Log

chkpt_StartCheckpoint()
    Advance Epoch
    Create checkpoint version and directory
    Save application state
    Save basic MPI state
        Number of nodes, local rank, local processor name
        Current epoch
        Attached buffers
    Save handle tables
        Includes datatypes and reduction operations
    Save and reset Early-Message-Registry
    Send Checkpoint-Initiated messages to every node Q with Sent-Count[Q]
    Prepare counters
        Copy Received-Counters to Late-Received-Counters
        Copy Early-Received-Counters to Received-Counters
        Reset Early-Received-Counters

chkpt_CommitCheckpoint()
    Save and reset Late-Message-Registry
    Commit Checkpoint to disk
    Close checkpoint

chkpt_RestoreCheckpoint()
    Initialize MPI
    Query last local saved checkpoint committed to disk
    Global reduction to find last checkpoint committed on all nodes
    Open Checkpoint
    Mode:=Restore

    Restore basic MPI state
        Number of nodes, local rank, local processor name
        Current epoch
        Attached buffers
    Restore handle tables
        Includes datatypes and reduction operations
    Restore message registries
        Restore Late-Message-Registry
        Restore Early-Message-Registry
        Distribute Early-Message-Registry entries to respective
        target nodes to form Was-Early-Registry
        Reset Early-Message-Registry

        If  (Late-Message-Registry is empty) and
            (Was-Early-Registry is empty)
                Mode:=Run

    ... jump to checkpointed location ...
    ... resume execution ...
```

Figure 5: Protocol actions

a signature in this registry is suppressed, and the signature is removed from the registry.

Similarly, if a message receive matches a message in the Late-Message-Registry, the data for that receive is received from this registry, and the entry for that message is removed from the registry. In addition, the signatures stored in the Late-Message-Registry are used to fill in any wild-cards to force intra-epoch messages to be received in the order that they were received prior to failure.

When the Was-Early-Registry and the Late-Message-Registry are empty, recovery is complete, and the process transitions to the Run state.

3.2 Piggybacked Information on Messages

Because MPI does not provide any FIFO guarantees for messages with different signatures, the protocol layer must piggyback a small amount of information on each application message to permit the receiver of a message to determine the state of the sending process at the time the message was sent. These piggybacked values are derived from the Epoch and Mode variables maintained by each process. The protocol layer piggybacks these values on all application messages. The receiver of the message uses this piggybacked information to answer the following questions.

1. Is the message a late, intra-epoch, or early message?
   This is determined by comparing the piggybacked epoch with the epoch that the receiving process is in, as described in Definition 1.
2. Has the sending process stopped logging non-determinstic events?
   No, if the piggybacked *mode* is **NonDet-Log**, and yes otherwise.

A detailed examination of the protocol shows that further economy in piggybacking can be achieved if we exploit the fact that a message can cross at most one recovery line. If we imagine that epochs are colored red, green, and blue successively, we see that the integer Epoch can be replaced by Epoch-color, which can be encoded in two bits. Furthermore, a single piggybacked bit is adequate to encode whether the sender of a message has stopped logging non-deterministic events. Therefore, it is sufficient to piggyback three bits on each outgoing message. For simplicity, we do not show these optimizations in the pseudo-code.

4. ADVANCED MPI FEATURES

The basic protocol described in Section 3 applies to all blocking point-to-point communication. In this section, we describe how we extend these mechanisms to implement advanced MPI features such as non-blocking communication, complex datatypes, and collectives.

4.1 Non-blocking Communication

MPI provides a set of routines to implement non-blocking communication, which separates the initiation of a communication call from its completion. The objective of this separation is to permit the programmer to hide the latency of the communication operation by performing other computations between the time the communication is initiated and the time it completes. MPI provides non-blocking send and receive calls to initiate communication, and it provides a variety of blocking wait or non-blocking test calls to determine completion of communication requests.

Extending the Basic Protocol

Non-blocking communication does not complete during a single call to the MPI library, but is active during a period of time between the initiation and the completion of the communication. During this time, MPI maintains a request object to identify the active communication. If a checkpoint is taken between the time the process initiates a non-blocking communication and the time that this communication completes, the protocol layer has to ensure that the request object is restored correctly during recovery.

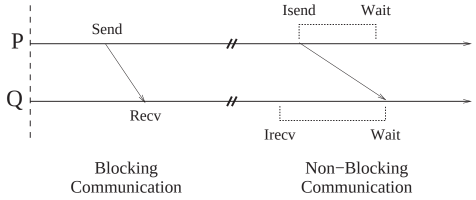

Figure 6: Mapping the base protocol onto non-blocking commination

To extend our protocol to non-blocking communication, non-blocking send operations execute the send protocol described in Section 3, while the receive protocol is executed within the Test and Wait calls. To handle Test and Wait calls, we must store additional information in the request objects created by the non-blocking receive calls. To stay independent of the underlying MPI implementation, we implement a separate indirection table for all requests. For each request allocated by MPI, we allocate an entry in this table and use it to store the necessary information, including type of operation, message parameters, and the epoch in which the request has been allocated. In addition, we store a pointer to the original MPI request object. The index to this table replaces the MPI request in the target application. This enables our MPI layer to instantiate all request objects with the same request identifiers during recovery.

At checkpoint time, the request table on each node contains all active requests crossing the recovery line and hence all requests that need to be restored during a restart from that recovery line. However, at this time we do not know which of the open receive requests will be completed by a late message. This is important, since late messages are replayed from the log during restart and hence should not be recreated. Therefore, we delay the saving of the request table until the end of the checkpoint period when all late messages have been received. During the logging phase, we mark the type of message matching the posted request during each completed Test or Wait call. In addition, to maintain all relevant requests, we delay any deallocation of requests until after the request table has been saved.

During recovery, all requests allocated during the logging phase, i.e., after the recovery line, are first deleted to roll the contents of the request table back from the end of the checkpoint period to the recovery line. Then, all requests that have not been completed by a late message are recreated before the program resumes execution.

Dealing with Nondeterminism

As described in Section 3, our protocol contains a phase that logs any potential non-determinism. For non-blocking communication, this has to include the correct recording of the number of unsuccessful tests as well as the logging of the completed indices in calls containing arrays of requests.

For this purpose, we maintain a test counter for each request to record the number of unsuccessful Test or Wait operations on this request. This counter is reset at the beginning of each checkpointing period and saved at the end of the checkpointing period as part of the request table. At recovery time, a Test call checks this counter to determine whether the same call during the original run was successful. If not, i.e., the counter is not zero, the counter is decremented and the call returns without attempting to complete the request. If, however, the original call was successful, i.e., the counter has reached zero, the call is substituted with a corresponding Wait operation. This ensures that the Test completes as in the original execution. Similarly, this counter is used to log the index or indices of MPI_Wait_any and MPI_Wait_some and to replay these routines during recovery.

Note that this replacement of Test calls with Wait calls can never lead to deadlock, since the Test completed during the original execution, and hence a corresponding message either has already arrived or is expected to arrive. The Wait is therefore guaranteed to complete during recovery.

4.2 Handles for Datatypes

MPI provides routines to define application-specific datatypes. These datatypes can then be used during communication requests to specify message payloads. To support datatypes in our protocol, we use an indirection table similar to the request table to store both the original MPI datatype handle and the information that was used during the creation of that datatype. During recovery, this information is used to recreate all datatypes before the execution of the program resumes.

This process is complicated by the fact that MPI datatypes can be constructed using other, previously constructed datatypes, resulting in a hierarchy of types. We keep track of this hierarchy within the datatype table by storing the indices of the dependent types with each entry. In addition, we ensure that datatypes are not actually deleted until both the datatype itself and all types depending on it have been deleted. This ensures that during a restore all intermediate datatypes have can be correctly reconstructed.

The information about the datatype hierarchy is also used for any message logging or restoration. This is necessary, since MPI datatypes can be represent non-contiguous memory regions. In both cases, the datatype hierarchy is recursively traversed to identify and individually store or retrieve each piece of the message.

4.3 Collective Communication

MPI offers a variety of collective communication primitives. The main problem with collective communication calls is that some processes may execute the call before taking their checkpoints while other processes may execute the call after taking their checkpoints. Unless something is done, only a subset of the processes will therefore participate in the collective communication call during recovery, which is erroneous.

Although we could convert each collective communication call to a set of point-to-point messages and apply the protocol described in Section 3 to these messages, we do not do this because it is important to permit the application to use the native, optimized collective calls.

Handling Collective Communication

The approach we take is similar to our approach for non-blocking communication calls in the sense that we apply the protocol only to the start and end points of each individual communication stream within a collective operation, without affecting the actual data transfer mechanisms in the underlying MPI layer.

We show an example of this approach in Figure 7. MPI_Gather aggregates data from all processors (marked as "In") into a single buffer on node (marked as "Out"). At the call site on each process, we first apply the send protocol shown in Figure 4. After the necessary protocol updates have been made, the protocol layer uses the original MPI_Gather operation to carry out the communication and thereby takes advantage of potential optimizations.

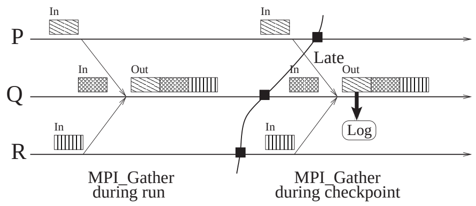

Figure 7: Example of collective communication: MPI_Gather

At the root node (in this case process Q), the protocol, after receiving the communication by calling MPI_Gather, performs counter and registry updates for each communication stream by applying the receive protocol described in Figure 4. This enables the protocol to determine if part of the communication is late or early and apply the necessary protocol mechanisms only to those message parts, as shown on the right of Figure 7 for the late communication from P to Q.

During recovery, we emulate collectives using point-to-point calls and apply the the protocol as describe above to each individual communication stream within each single collective operation. Once the complete application has recovered, we switch back to using the original collective calls. Therefore, any performance penalty is restricted to the short phase of process recovery.

Reduction Operations

The approach described above cannot be applied directly to reduction operations, such as MPI_Reduce, MPI_Allreduce, and MPI_Scan. These routines aggregate the message payload based on a user-provided reduction operation before returning the data to the user. Hence, it is no longer possible to log individual messages, which is a requirement of the base protocol.

For MPI_Allreduce however, it is sufficient to store the final result of the operation at each node and replay this from the log during recovery. This operation involves an all-to-all communication scheme, and as a consequence, each communication process will have outstanding early and late messages during the MPI_Allreduce call. Hence, all communication is completed either before a checkpoint or during nondeterministic logging. This ensures that the complete collective operation is repeated exactly as during the original run and provides the same result.

Similarly MPI_Scan can be implemented by logging the result of the routine. The use of the prefix operator results in a strictly ordered dependency chain between processes. This guarantees that communication for MPI_Scan is crossed at most once by a recovery line. At recovery, the first process to call MPI_Scan is restored from a logged value, while all following processes can be recovered by recomputing along this dependency chain based on the logged data.

In contrast to these two routines, MPI_Reduce does not have similar properties and hence parts of the communication contributing to the final result can be intra-epoch messages. The payload of these messages can change during a restore, and, as a consequence, simply logging the final result and replaying this during recovery is insufficient. To compensate for this behavior, we first send all data to the root node of the reduction using an independent `MPI_Gather` and then perform the actual reduction. This provides the protocol with the required individual messages from all processes and allows a correct replay on recovery.

4.4 Communicators, Groups, and Topologies

Our protocol layer currently does not support arbitrary communicators, groups, and topologies. An extension providing such support, however, is straightforward and is currently under development. Similarly to datatypes, any creation or deletion has to be recorded and stored as part of the checkpoint. On recovery, we read this information and replay the necessary MPI calls to recreate the respective structures.

5. STATE SAVING

The protocol described in Sections 3 and 4 is independent of the way in which the local computational state is saved by each process. For completeness, we describe the implementation of state-saving in our system.

Roughly speaking, the state of an individual process consists both of its data and its execution context. To provide CPR, the C^3 system utilizes a pre-compiler to instrument certain points in an application's code where there are changes to either its set of variables (and objects) or to its execution context.

This inserted code usually consists of a call to a routine in the C^3 utility library that registers the change to the application's state. For example, as a local variable enters scope, the inserted code passes a description of that variable to the utility library, where it is added to the set of variables in scope. When it leaves scope, the variable's description is removed from that set. In this manner, the inserted code, when executed dynamically, serves to maintain an up-to-date description of the processes' state.

As mentioned before, the C^3 system requires that the programmer specify the positions in the application where CPR may occur, by marking them with a `#pragma` statement. Because the set of such locations is necessarily finite, the pre-compiler only needs to instrument the code when changes to the application's state cross such a position.

When it is time to take a checkpoint, the C^3 system uses the description of the process state that it had maintained to write the state to the checkpoint file. It then stores the description to the checkpoint file as well. When restarting, first the description is read, and then it is used to reconstruct the application's state from the information saved within the checkpoint file.

Although the checkpointing mechanism used by C^3 is portable, the checkpoints are not: C^3 saves all data as binary, irrespective of the data's type. This was the result of a design philosophy that favors efficiency (not needing to convert data to a neutral format) and transparency (not confining programmers to a subset of C with limited pointers) to portability. Because all data is saved as binary, on restart, the C^3 system must ensure that all objects and variables are restored to their original addresses, otherwise pointers would no longer be correct after a restart. For stack allocated variables, this is accomplished by "padding" the stack before handing control to `main`. For dynamically allocated objects, C^3 provides its own memory manager.

6. PERFORMANCE

To evaluate the quality of C^3, we would have liked to compare its performance with that of a more established system. However, there is no other parallel checkpointing system that is available for our target platforms. Therefore, we performed experiments to answer the following questions.

- How do checkpoint files produced by C^3 compare in size with those produced by other systems (on sequential processors)?
- How much overhead does the C^3 system add to an application when no checkpoints are taken?
- How much overhead does the C^3 system add to an application when checkpoints are taken?

We focused on the NAS Parallel Benchmarks (NPB), which are interesting to us because, with the exception of the MG benchmark, they do not contain calls to `MPI_Barrier` in the computations. Several of the codes call `MPI_Barrier` immediately before starting and stopping the benchmark timer, but only MG calls `MPI_Barrier` during the computation.

The experimental results that we present below are preliminary and they will be significantly expanded in the final version of the paper.

6.1 Checkpoint Sizes

To evaluate the checkpoint sizes of C^3, we compared the sizes of the checkpoint files produced by C^3 and Condor [14], arguably the most popular SLC system in high-performance computing. Since Condor only checkpoints sequential applications, we have measured the checkpoint sizes produced on uniprocessors.[1]

Table 1 shows the sizes of the checkpoint files produced by C^3 and Condor for the NAS Parallel Benchmarks (NPB's) for both Solaris and Linux.

Solaris A SUN V210 with a two 1GHz UltraSPARC IIIi processors, 1 MB L2 cache, and 2 GB RAM, running Solaris 9.

Linux A Dell PowerEdge 1650, with a 1.26GHz Intel Pentium III processor, 512K L2 cache, and 512MB of RAM, running Redhat Linux 8.0.

			Size in megabytes		
O.S.	Bench	Class	Condor	C^3	Reduction
Solaris	BT	A	308.85	306.39	0.80%
	CG	B	429.89	427.44	0.57%
	EP	A	3.46	1.00	71.07%
	FT	A	421.28	418.69	0.61%
	IS	A	100.45	96.00	4.43%
	LU	A	46.99	44.54	5.21%
	MG	B	436.99	435.48	0.34%
	SP	A	82.09	79.63	2.99%
Linux	BT	A	307.13	306.39	0.24%
	CG	B	428.17	427.44	0.17%
	EP	A	1.74	1.00	42.29%
	FT	A	419.43	418.69	0.17%
	IS	A	96.74	96.00	0.76%
	LU	A	45.27	44.54	1.61%
	MG	B	435.24	435.55	-0.07%
	SP	A	80.36	79.63	0.91%

Table 1: Condor and C^3 checkpoint sizes

The sizes of the checkpoint files are given in megabytes, and the column labeled "Reduction" is the relative amount that the C^3 checkpoints are smaller than the Condor checkpoiunts.

[1] CoCheck [18] is a SLC system based on Condor for MPI applications, but it does not run on any of our target platforms.

These results show that in almost all cases, the checkpoints produced by the C^3 system are smaller than those produced by Condor. This is primarily because the C^3 system saves only live data (memory that has not been freed by the programmer) from the heap. Because C^3 is an ALC system, the checkpoint files can be further reduced by applying compiler analysis and optimizations. This is not possible with an SLC system like Condor.

6.2 Overhead Without Checkpoints

To measure the overhead of using the fault-tolerance protocol described in Section 3, we ran experiments on two high-performance parallel machines.

Velocity 2 The Velocity 2 cluster at the Cornell Theory Center consists of 128 dual processor 2.4GHz Intel P4 Xeon nodes. Each processor has a 512KB L2 cache. and runs Windows Advanced Server 2000. Each node has 2 GB of RAM and a 72GB local disk. The nodes are connected with Force10 Gigabit Ethernet.

Lemieux The Lemieux system at the Pittsburgh Supercomputing Center consists of 750 Compaq Alphaserver ES45 nodes. Each node contains four 1-GHz Alpha processors and runs the Tru64 Unix operating system. Each node has 4 Gbytes of memory and 38G local disk. The nodes are connected with a Quadrics interconnection network.

Tables 2 and 3 show the running times of some of the NPB's for Velocity 2 and Lemieux, respectively. We were not able to get sufficient time on these machines to complete all the NPB's, so we show performance results for only the benchmarks we were able to run. The column labeled "Original" shows the running time in seconds of the original benchmark application. The column labeled "C^3" shows the running time of the application that has been compiled and run using the C^3 system. For these runs, no checkpoints are taken. The column labeled "Relative" shows the relative overhead of using the C^3 system. This overhead comes from executing the book-keeping code inserted by the pre-compiler, and the piggybacking and bookkeeping done by our MPI protocol layer.

Code	Class	Procs	Original	C^3	
		(Nodes)	Runtime	Runtime	Relative
CG	D	128 (64)	1746.06	1787.54	2.38%
SP	D	81 (81)	2406.00	2517.75	4.64%
		144 (72)	2114.86	2218.38	4.89%
LU	D	64 (64)	2092.50	2114.38	1.05%
		128 (64)	1861.97	1904.56	2.29%

Table 2: Runtimes on Velocity 2 without checkpoints

Code	Class	Procs	Original	C^3	
		(Nodes)	Runtime	Runtime	Relative
LU	D	64 (16)	1490.87	1707.78	14.55%
		256 (64)	396.35	466.37	17.67%
		1024 (256)	128.54	145.69	13.34%

Table 3: Runtimes on Lemieux without checkpoints

The overheads measured on Lemieux are higher than we would like. Our preliminary studies show that these overheads are present in the sequential code as well, which suggests that the performance problems are present in the state-saving mechanism and not in the protocol layer. We are investigating this further, and we will report on the improved results in the final paper.

Nevertheless, we are encouraged by these results because they show that the overhead added by using C^3 does not grow significantly as the number of processors in increased. Furthermore, the overheads measured on Velocity 2 are small and less than 5% in all cases. These results suggest that the protocol is scalable.

6.3 Overhead With Checkpoints

Our final set of experiments are designed to measure the additional overhead of taking checkpoints.

Tables 4 and 5 show the run-times and absolute overheads in seconds of taking checkpoints for the same applications shown in Tables 2 and 3. The meaning of the configurations is as follows.

Configuration #1. The run-times of the C^3 generated code without taking any checkpoints. These run-times are the same as shown in column "C^3" of Tables 2 and 3, and serve as the baseline for measuring the overhead of checkpointing.

Configuration #2. The run-times of the C^3 generated code when computing one checkpoint during the application run but without saving any checkpoint data to disk.

Configuration #3. This configuration is the same as #2, except that it includes the cost of saving application state to the local disk on each node.

The difference between Configurations #2 and #3 is that #3 includes the cost of the parallel checkpointing protocol (i.e., constructing the recovery line, logging late messages, determining the early messages, and termination) and the cost of saving the application state to local disk, while #2 includes only the cost of the parallel checkpointing protocol.

These results show that the cost of taking a checkpoint is small. To put these results into perspective, if we scale the running times appropriately, then we see that the maximum overhead when checkpointing once an hour is 1.11% and the maximum overhead when checkpointing once a day is 0.05%.

In some cases, taking checkpoints actually improves the performance of the fault-tolerant application. One possible explanation for this is that the overheads that we are measuring are within range of measurement error. If this is the case, then the cost of checkpointing these applications is essentially zero. Further experiments will determine whether or not this is the case.

As we mentioned earlier, Configuration #3 measures the cost of writing the application state to each node's local disk. In a production system, writing checkpoint files to local disk does not ensure fault-tolerance, because when a node is inaccessible, its local disk usually is too. However, writing directly to a non-local disk is usually not a good idea because the network contention and communication to off-cluster resources can add significant overhead. A better strategy that is used by some systems [19] is for the application to write checkpoints to a local disk and then for an external daemon to asynchronously transfer these checkpoints from local disk to an off-cluster disk. Very often a second, possibly lower performance, network is used to avoid contention with the application's messages.

At present, we have not implemented such a daemon, so we cannot measure its cost. However, it is worth noting that any parallel checkpointing system will require such a system, so this is a source of overhead that will be present in all these systems.

6.4 Discussion

			Configurations				
Code	Class	Procs	#1	#2		#3	
		(Nodes)	Runtime	Runtime	Absolute	Runtime	Absolute
CG	D	128 (64)	1787.54	1806.51	18.97	1827.19	39.65
SP	D	81 (81)	2517.75	2519.76	2.01	2526.08	8.33
		144 (72)	2218.38	–*	–*	2258.38	40.00
LU	D	64 (64)	2114.38	2136.39	22.01	2125.26	10.88
		128 (64)	1904.56	1875.81	-28.75	1861.70	-42.86

*These results were unavailable at the time of submission

Table 4: Runtimes on Velocity 2 with checkpoints

			Configurations				
Code	Class	Procs	#1	#2		#3	
		(Nodes)	Runtime	Runtime	Absolute	Runtime	Absolute
LU	D	64 (??)	1707.78	1705.21	-2.57	1722.81	15.03
		256 (??)	466.37	461.44	-4.93	471.87	5.50
		1024 (256)	145.69	147.24	1.55	152.24	6.55

Table 5: Runtimes on Lemieux with checkpoints

The experimental results reported here are preliminary and for the final version of this paper, we plan to include results from additional, larger applications. While the C^3 system has been successfully used to checkpoint and restart sequential versions of all of the NAS codes, as reported in Table 1, we are still debugging some of the parallel versions. We will report on all of the parallel performance of all of the NAS codes in the final version.

Also, we have used our system to add checkpointing to the SMG2000 code from the ASCI Purple benchmarks [6]. The resulting code runs successfully, but we have not reported its results because we are still debugging its restart.

For the final version of this paper, we also plan on reporting results for High Performance LINPACK [15]. HPL is the benchmark used for computing performance for the Top500 Supercomputer Sites and is extremely scalable.

7. RELATED WORK

While much theoretical work has been done in the field of distributed fault-tolerance, few systems have been implemented for actual distributed application environments.

High-availability Systems In the distributed systems community, fault-tolerance has been studied mainly in the context of ensuring zero downtime for critical systems such as web-servers and air-traffic controller systems [13]. The problem of tolerating faults in the context of high-performance computing is fundamentally different in nature because the objective is to minimize the expected time to completion of a program, given some probability of failure. Distributed systems techniques, such as fail-over or replication of computations, are not useful in this context because they reduce the resources available to the computation between failures. Alvisi et al [9] is an excellent survey of techniques developed by the distributed systems community for recovering from fail-stop faults.

System-level Checkpointing Condor is used widely for sequential system-level checkpointing on Unix systems [14]. The CoCheck system [18] provides the functionality for the coordination of distributed checkpoints, relying on Condor to take system-level checkpoints of each process. In contrast to our approach, CoCheck is integrated with its own MPI implementation, and assumes that collective communications are implemented as point-to-point messages. We believe that our ability to inter-operate with any MPI implementation is a significant advantage. A blocking coordinated system-level checkpointing solution is described in [19].

Message-logging There is an entire class of recovery protocols called *message-logging protocols*, of which the Manetho system [8] is an exemplar. In message-logging, processes that survive a hardware failure are not rolled back; instead, only the failed processes are restarted, and surviving processes help them recover by replaying messages they sent to the restarted processes before failure.

Manetho uses an approach called causal message-logging. Because a Manetho process logs both the data of the messages that it sends and the non-deterministic events that these messages depend on, the size of those logs may grow very large if used with a program that generates a high volume of large messages, as is the case for most scientific programs. While Manetho can bound the size of these logs by occasionally checkpointing process state to disk, programs that perform a large amount of communication would require very frequent checkpointing to avoid running out of log space. Furthermore, since the system requires a process to take a checkpoint whenever these logs get too large, it is not clear how to use this approach in the context of application-level checkpointing. Note that although our protocol, like the Chandy-Lamport protocol, also records message data, recording happens only during checkpointing, and not during normal execution. Another difference is that Manetho was not designed to work with any standard message passing API, and thus does not need to deal with the complex constructs – such as non-blocking and collective communication – found in MPI.

The Egida [17] system is another fault-tolerant system for MPI. Like Manetho, it is primarily based upon message logging, and uses checkpointing to flush the logs when they grow too large.

Manual Application-level Checkpointing Several systems have been developed to make ALC easier to program. The Dome (Distributed Object Migration Environment) system [3] is a C++ library based on data-parallel objects. SRS [20] allows the programmer to manually specify the data that needs to be saved as well as its distribution. On recovery the system uses this information to recover the program's state and redistribute the data on a potentially different number of processors.

Automatic Sequential Application-level Checkpointing Porch [16] supports portable ALC for programs written in a restricted subset of C. It generates runtime meta-information that provides size and alignment information for basic types and layout information, which allows the checkpointer to convert all data to a universal checkpoint format. The APrIL system [10] uses techniques similar to Porch, but uses heuristic techniques for determining the type of heap objects.

Reducing Checkpoint Size Beck and Plank [2] used a context-insensitive live variable analysis to reduce the amount of state information that must be saved when checkpointing. The CATCH [12] system uses profiling to determine the likely size of the checkpoints at different points in the program. A learning algorithm is then used to choose the points at which checkpoints should be taken so that the size of the saved state is minimized while keeping the checkpoint interval optimal.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have shown that application-level non-blocking coordinated checkpointing can be used to add fault-tolerance to C/MPI programs. We have argued that existing checkpointing protocols are not adequate for this purpose and we have developed a novel protocol to meet the need.

We have presented a system that can be used to transform C/MPI programs to use our protocol. This system uses program transformation technology to transform the application so that it will save and restore its own state. We have shown how the state of the underlying MPI library can be reconstructed by the implementation of our protocol.

The protocols presented in this paper offer significant improvements and enhancements to the protocols presented in [4] and [5]. These changes came about as a result of our first complete implementation of these protocols. The performance results presented in this paper show that our implementation is able to deliver scalable performance on two very different state-of-the-are supercomputing systems.

The ultimate goal of our project is to provide a highly efficient checkpointing mechanism for MPI applications. One way to minimize checkpoint overhead is to reduce the amount of data that must be saved when taking a checkpoint. Previous work in the compiler literature has looked at analysis techniques for avoiding the checkpointing of dead and read-only variables [2]. This work focused on statically allocated data structures in FORTRAN programs. We would like to extend this work to handle the dynamically created memory objects in C/MPI applications. We are also studying incremental checkpointing approaches for reducing the amount of saved state.

Another powerful optimization is to trade off state-saving for recomputation. In many applications, the state of the entire computation at a global checkpoint can be recovered from a small subset of the saved state in that checkpoint. The simplest example of this optimization is provided by a computation in which we need to save two variables x and y. If y is some simple function of x, it is sufficient to save x, and recompute the value of y during recovery, thereby trading off the cost of saving variable y against the cost of recomputing y during recovery. Real codes provide many opportunities for applying this optimization. For example, in protein-folding using *ab initio* methods, it is sufficient to save the positions and velocities of the bases in the protein at the end of a time-step because the state of the entire computation can be recovered from that data.

Acknowledgements: Some of the experiments reported in this paper were performed on the National Science Foundation Terascale Computing System at the Pittsburgh Supercomputing Center, while other experiments were performed on the Velocity cluster in the Cornell Theory Center.

9. REFERENCES

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[2] M. Beck, J. S. Plank, and G. Kingsley. Compiler-assisted checkpointing. Technical Report UT-CS-94-269, Dept. of Computer Science, University of Tennessee, 1994.

[3] A. Beguelin, E. Seligman, and P. Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[4] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. In *Principles and Practices of Parallel Programming*, San Diego, CA, June 2003.

[5] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in an application-level fault tolerant MPI system. In *International Conference on Supercomputing (ICS) 2003*, San Francisco, CA, June 23–26 2003.

[6] B. Carnes. The smg2000 benchmark code. Available at `http://www.llnl.gov/asci/purple/benchmarks/limit` September 19 2001.

[7] M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

[8] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output. *IEEE Transactions on Computers*, 41(5), May 1992.

[9] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, Oct. 1996.

[10] A. J. Ferrari, S. J. Chapin, and 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, 25, 1997.

[11] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[12] C.-C. J. Li and W. K. Fuchs. Catch – compiler-assisted techniques for checkpointing. In *20th International Symposium on Fault Tolerant Computing*, pages 74–81, 1990.

[13] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, California, first edition, 1996.

[14] J. B. M. Litzkow, T. Tannenbaum and M. Livny. Checkpoint and migration of UNIX processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, 1997.

[15] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl - a portable implementation of the high-performance linpack benchmark for distributed-memory computers. Available at `http://www.netlib.org/benchmark/hpl/`.

[16] B. Ramkumar and V. Strumpen. Portable checkpointing for heterogenous architectures. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.

[17] S. Rao, L. Alvisi, and H. M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Symposium on Fault-Tolerant Computing*, pages 48–55, 1999.

[18] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[19] N. Stone, J. Kochmar, R. Reddy, J. R. Scott, J. Sommerfield, and C. Vizino. A checkpoint and recovery system for the Pittsburgh Supercomputing Center Terascale Computing System. In *Supercomputing*, 2001. Available at `http://www.psc.edu/publications/tech_reports/chk`

[20] S. Vadhiyar and J. Dongarra. Srs - a framework for

developing malleable and migratable parallel software.
Parallel Processing Letters, 13(2):291–312, June 2003.