C^3: A System for Automating Application-level Checkpointing of MPI Programs

Greg Bronevetsky, Daniel Marques, Keshav Pingali, Paul Stodghill *

Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract. Fault-tolerance is becoming necessary on high-performance platforms. Checkpointing techniques make programs fault-tolerant by saving their state periodically and restoring this state after failure. *System-level* checkpointing saves the state of the entire machine on stable storage, but this usually has too much overhead. In practice, programmers do manual *application-level* checkpointing by writing code to (i) save the values of key program variables at critical points in the program, and (ii) restore the entire computational state from these values during recovery. However, this can be difficult to do in general MPI programs.

In ([2],[3]) we have presented a distributed checkpoint coordination protocol which handles MPI's point-to-point and collective constructs, while dealing with the unique challenges of application-level checkpointing. We have implemented our protocols as part of a thin software layer that sits between the application program and the MPI library, so it does not require any modifications to the MPI library. This thin layer is used by the C^3 (Cornell Checkpoint (pre-)Compiler), a tool that automatically converts an MPI application in an equivalent fault-tolerant version. In this paper, we summarize our work on this system to date. We also present experimental results that show that the overhead introduced by the protocols are small. We also discuss a number of future areas of research.

1 Introduction

The problem of implementing software systems that can tolerate hardware failures has been studied extensively by the distributed systems community [6]. In contrast, the parallel computing community has largely ignored this problem 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, trends in high-performance computing, such as the popularity of custom-assembled clusters, the increasing complexity of parallel machines, and the dawn of Grid computing, are increasing the probability of hardware failures, making it imperative that parallel programs tolerate such failures.

One solution that has been employed successfully for parallel programs is application-level checkpointing. In this approach, the programmer is responsible for saving computational state periodically, and for restoring this state after failure. In many programs,

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

it is possible to recover the full computational state from relatively small amounts of data saved at key places in the program. For example, in an *ab initio* protein-folding application, it is sufficient to periodically save the positions and velocities of the bases of the protein; this is a few megabytes of information, in contrast to the hundreds of gigabytes of information that would be saved by a system-level checkpoint.

This kind of manual application-level checkpointing is feasible if the parallel program is written in a bulk-synchronous manner, but it is not clear how it can be applied to a general MIMD program without global 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. Protocols such as the Chandy-Lamport [4] protocol have been designed by the distributed systems community to address this problem, but these protocols were designed for system-level checkpointing, and cannot be applied to application-level checkpointing, as we explain in Section 4.

In two previous papers ([2],[3]), we have present *non-blocking, coordinated, application-level checkpointing* protocols for the point-to-point and collective constructs of MPI. We have implemented these protocols as part of the C^3 (Cornell Checkpoint (pre-)Compiler), a system that uses program transformation technology to automatically insert application-level checkpointing features into an application's source code. Using our system, it is possible to automatically convert an MPI application in an equivalent fault-tolerant version.

The rest of this paper is organized as follows. In Section 2, we present background for and define the problem. In Section 3, we define some terminology and describe our basic approach. In Section 4, we discuss some of the difficulties of adding fault-tolerance to MPI programs. In Sections 5 and 6 we present non-blocking checkpointing protocols for point-to-point and collective communication, respectively. In Section 7, we discuss how our system saves the sequential state of each process. In Section 8, we present performance results of our system. In Section 9 we discuss related work, and in Section 10 we describe future work. In Section 11, we offer some conclusions.

2 Background

To address the problem of fault tolerance, it is necessary to define the fault model. We focus our attention on stopping faults, in which a faulty process hangs and stops responding to the rest of the system, neither sending nor receiving messages. This model captures many failures that occur in practice and is a useful mechanism in addressing more general problems.

We make the standard assumption that there is a reliable transport layer for delivering application messages, and we build our solutions on top of that abstraction. One such reliable implementation of the MPI communication library is Los Alamos MPI [7].

We can now state the problem we address in this paper. We are given a long-running MPI program that must run on a machine that has (i) a reliable message delivery system, (ii) unreliable processors which can fail silently at any time, and (iii) a mechanism such as a distributed failure detector [8] for detecting failed processes. How do we ensure that the program makes progress in spite of these faults?

There are two basic approaches to providing fault-tolerance for distributed applications. *Message-logging* techniques require restarting only the computation performed by the failed process. Surviving processes are not rolled back but must help the restarted process by replaying messages that were sent to it before it failed. Our experience is that the overhead of saving or regenerating messages tends to be so overwhelming that the technique is not practical for scientific applications. Therefore, we focus on *Checkpointing* techniques, which periodically save a description of the state of a computation to stable storage; if any process fails, all processes are rolled back to a previously saved checkpoint (not necessarily the last), and the computation is restarted from there.

Checkpointing techniques can be classified along two independent dimensions.

(1) The first dimension is the abstraction level at which the state of a process is saved. In *system-level checkpointing* (e.g., [9], [11]), the raw process state, including the contents of the program counter, registers and memory, are saved on stable storage. Unfortunately, complete system-level checkpointing of parallel machines with thousands of processors can be impractical because each global checkpoint can require saving terabytes of data to stable storage. For this reason, system-level checkpointing is not done on large machines such as the IBM Blue Gene or the ASCI machines.

One alternative which is popular is *application-level checkpointing*, in which the application is written such that it correctly restarts from various positions in the code by storing certain information to a restart file. The benefit of this technique is that that the programmer needs to save only the minimum amount of data necessary to recover the program state. In this paper, we explore the use of compiler technology to automate application-level checkpointing.

(2) The second dimension along which checkpointing techniques can be classified is the technique used to coordinate parallel processes when checkpoints need to be taken. In [2], we argue that the best approach for our problem is to use *non-blocking coordinated* checkpointing. This means that all of the processes participate in taking each checkpoint, but they do not stop the computation while they do so. A survey of the other approaches to checkpointing can be found in [6].

3 Our Approach

3.1 Terminology

We assume that a distinguished process called the *initiator* triggers the creation of global checkpoints periodically. We assume that it does not initiate the creation of a global checkpoint before any previous global checkpoint has been created and committed to stable storage.

The execution of an application process can therefore be divided into a succession of *epochs* where an epoch is the period between two successive local checkpoints (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 1.

Application messages can be classified depending upon whether or not they are sent and received in the same epoch.

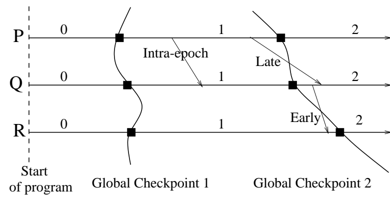
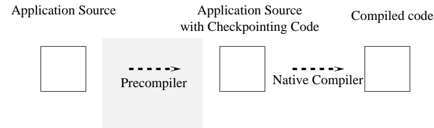


Fig. 1. Epochs and message classification

Compile Time



Run Time

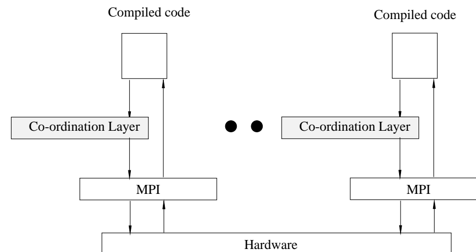


Fig. 2. System Architecture

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 1 shows examples of the three kinds of messages, using the execution trace of three processes named P, Q and R. MPI has several kinds of send and receive commands, so it is important to understand what the message arrows mean in the context of MPI programs. Consider the late message in Figure 1. 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. Note that if this routine is a non-blocking send, the message may not make it to the communication network until much later in execution; nevertheless, what is important for us is that if the application tries to recover from global checkpoint 2, it will not reissue the MPI send. Similarly, the destination of the arrow represents the delivery of the message to the application program. In particular, if an `MPI_Irecv` is used by the receiving process to get the message, the destination of the arrow represents the point at which an `MPI_Wait` for the message would have returned, not the point where control returns from the `MPI_Irecv` routine.

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 but in our opinion, it is misleading in the context of application-level checkpointing.

3.2 System Architecture

Figure 2 is an overview of our approach. The C^3 system reads almost unmodified single-threaded C/MPI source files and instruments them to perform application-level state-saving; the only additional requirement for the programmer is that he insert calls to a function called `PotentialCheckpoint` at points in the application where the programmer wants checkpointing to occur. 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. Further, it allows us to easily migrate from one MPI implementation to another.

4 Difficulties in Application-level Checkpointing of MPI programs

In this section, we briefly describe the difficulties with implementing application-level, coordinated, non-blocking checkpointing for MPI programs.

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 a program executes a* `PotentialCheckpoint` *call.*

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. This strategy does not work for application-level checkpointing, because, after being notified to take a checkpoint, a process might need to communicate with other processes before arriving at a point where it may take a checkpoint.

Handling late and early messages Suppose that an application is restored to Global Checkpoint 2 in Figure 1. On restart, some processes will expect to receive *late* messages that were sent prior to 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. Late messages must be handled by non-blocking system-level checkpointing protocols as well.

Similarly on recovery, some processes will expect to send *early* messages that were received prior to failure. To handle this, we need mechanisms for (i) identifying early messages, and (ii) ensuring that they are not resent during recovery.

Early messages also pose a separate and more subtle problem: if a non-deterministic event occurs between a checkpoint and an early message send, then on restart the event may occur difference and, hence, the message may be different. In general, we must ensure that if a global checkpoint depends on a non-deterministic event, that the event will re-occur exactly the same way after restart. 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.

Non-FIFO message delivery at application level In an MPI application, a process P can use tag matching to receive messages from Q in a different order than they were sent. Therefore, a protocol that works at the application-level, as would be the case for application-level checkpointing, cannot assume FIFO communication.

Collective communication The MPI standard includes collective communications functions such as `MPI_Bcast` and `MPI_Alltoall`, which involve the exchange of data among a number of processors. The difficulty presented by such functions occurs when some processes make a collective communication call before taking their checkpoints, and others after. We need to ensure that on restart, the processes that reexecute the calls do not deadlock or receive incorrect information. Furthermore, `MPI_Barrier` guarantees specific synchronization semantics, which must be preserved on restart.

Problems Checkpointing MPI Library State The entire state of the MPI library is not exposed to the application program. Things like the contents of message buffers and request objects are not directly accessible. Our system must be able to reconstruct this hidden state on recovery.

5 Protocol for point-to-point operations

We now sketch the coordination protocol for global checkpointing for point-to-point communication. A complete description of the protocol can be found in [2].

5.1 High-level description of protocol

Initiation As with other non-blocking coordinated checkpointing protocols, we assume the existence of an *initiator* that is responsible for deciding when the checkpointing process should begin. In our system, the processor with rank 0 in `MPI_COMM_WORLD` serves as the initiator, and starts the protocol when a certain amount of time has elapsed since the last checkpoint was taken.

Phase #1 The initiator sends a control message called *pleaseCheckpoint* to all application processes. After receiving this message, each process can send and receive messages normally.

Phase #2 When an application process reaches its next `potentialCheckpoint` location, it takes a local checkpoint using the techniques described in Section 7. It also saves the identities of any early messages on stable storage. It then starts recording (i) every late message it receives, and (ii) the result of every non-deterministic decision it makes. Once a process has received all of its late messages[1], it sends a control message called *readyToStopRecording* back to the initiator, but continues recording.

[1] We assume the application code receives all messages that it sends.

Phase #3 When the initiator gets a *readyToStopRecording* message from all processes, it sends a control message called *stopRecording* to all other processes.

Phase #4 An application process stops recording when (i) it receives a *stopRecording* message from the initiator, or (ii) it receives a message from a process that has stopped its recording.

The second condition is required because we make no assumptions about message delivery order. In particular, it is possible for a recording process to receive a message from non-recording process before receiving the *stopRecording* message. In this case, the saved state might depend upon an unrecorded non-deterministic event. The second condition prevents this situation from occurring.

Once the process has saved its record on disk, it sends a *stoppedRecording* message back to the initiator. When the initiator receives a *stoppedRecording* message from all processes, it commits the checkpoint that was just created as the one to be used for recovery, saves this decision on stable storage, and terminates the protocol.

5.2 Piggybacked information on messages

To implement this protocol, the protocol layer must piggyback a small amount of information on each application message. The receiver of a message uses this piggybacked information to answer the following questions.

1. Is the message a late, intra-epoch, or early message?
2. Has the sending process stopped recording?
3. Which messages should not be resent during recovery?

The piggybacked values on a message are derived from the following values maintained on each process by the protocol layer.

- `epoch`: This integer keeps track of the process epoch. It is initialized to 0 at start of execution, and incremented whenever that process takes a local checkpoint.
- `amRecording`: This boolean is true when the process is recording, and false otherwise.
- `nextMessageID`: This integer is initialized to 0 at the beginning of each epoch, and is incremented whenever the process sends a message. Piggybacking this value on each application message in an epoch ensures that each message sent by a given process in a particular epoch has a unique ID.

A simple implementation of the protocol can piggyback all three values on each message that is sent by the application. When a message is received, the protocol layer at the receiver examines the piggybacked epoch number and compares it with the epoch number of the receiver to determine if the message is late, intra-epoch, or early. By looking at the piggybacked boolean, it determines whether the sender is still recording. Finally, if the message is an early message, the receiver adds the pair <sender, messageID> to its `suppressList`. Each process saves its `suppressList` to stable storage when it takes its local checkpoint. During recovery, each process passes relevant portions of its list of messageID's to other processes so that resending of these messages can be suppressed.

By exploiting properties of the protocol, the size of the piggybacked information can be reduces to two booleans and an integer. By exploiting the semantics of MPI message tags, it is possible to eliminate the integer altogether, and piggyback only two boolean values, one to represent `epoch` and the other `amRecording`.

5.3 Completing the reception of late messages

Finally, we need a mechanism for allowing an application process in one epoch to determine when it has received all the late messages sent in the previous epoch.

The solution we have implemented is straight-forward. In every epoch, each process P remembers how many messages it sent to every other process Q (call this value $sendCount(P \rightarrow Q)$). Each process Q also remembers how many messages it received from every other process P (call this value $receiveCount(Q \leftarrow P)$). When a process P takes its local checkpoint, it sends a $mySendCount$ message to the other processes, which contains the number of messages it sent to them in the previous epoch. When process Q receives this control message, it can compare the value with $receiveCount(Q \leftarrow P)$ to determine how many more messages to wait for.

Since the value of $sendCount(P \rightarrow Q)$ is itself sent in a control message, how does Q know how many of these control messages it should wait for? A simple solution is for each process to send its $sendCount$ to every other process in the system. This solution works, but requires quadratic communication. More efficient solutions can be obtained by requiring processes that communicate with one another to explicitly open and close communication "channels".

5.4 Guarantees provided by the protocol

It can be shown that this protocol provides the following guarantees that are useful for reasoning about correctness.

Claim.

1. No process stops recording until all processes have taken their local checkpoints.
2. A process that has stopped recording cannot receive a late message. In Figure 3, this means that a message of the form b1 $\rightarrow$ g3 cannot occur.
3. A message sent by a process after it has stopped recording can only be received by a process that has itself stopped recording. In Figure 3, this means that messages of the form b3 $\rightarrow$ g2 or b3 $\rightarrow$ g1 cannot occur.

Figure 3 shows the possible communication patterns, given these guarantees.

6 Protocol for collective operations

In this section, we build on the mechanisms of the point-to-point protocol in order to implement a protocol for collective communication. A complete description of our protocols can found in [3].

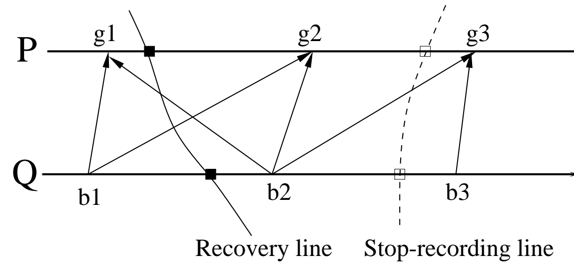


Fig. 3. Possible Patterns of Communication

There are two basic approaches to handling MPI's collective communication functions. The most obvious is to implement these functions on top of our point-to-point protocol. However, because this approach does not use the low-level network layer directly, it is likely to be less efficient than the collective functions provided by the native MPI library.

Instead, what we have chosen to do is to use the basic concepts and mechanisms of our point-to-point protocol in order to provide fault-tolerant versions of the collective communication functions *that are implemented entirely in terms of the native MPI collective communication functions*.

We will use `MPI_Allreduce` to illustrate how collective communication is handled. In Figure 4, collective communication call A shows an `MPI_Allreduce` call in which processes P and Q execute the call after taking local checkpoints, and process R executes the call before taking the checkpoint. During recovery, processes P and Q will reexecute this collective communication call, but process R will not. Unless something is done, the program will not recover correctly.

Our solution is to use the record to save the result of the `MPI_Allreduce` call at processes P and Q. During recovery, when the processes reexecute the collective communication call, the result is read from the record and returned to the application program. Process R does not reexecute the collective communication call. To make this intuitive idea precise, we need to specify when the result of a collective communication call like `MPI_Allreduce` should be recorded.

A simple solution is to require a process to record the result of every collective communication call it makes during the time it is recording. Collective communication call B in Figure 4 illustrates a subtle problem with this solution - process R executes the `MPI_Allreduce` after it has stopped recording, so it would be incorrect for processes P and Q to record the results of their call. This problem is similar to the problem encountered in the point-to-point message case, and the solution is similar (and simpler). Each process piggybacks its *amRecording* bit on the application data, and the function invoked by `MPI_Allreduce` computes the conjunction of these bits. If any process involved in the collective communication call has stopped recording, all the other processes will learn this fact, and they will also stop recording. As a result, no process will record the result of the call.

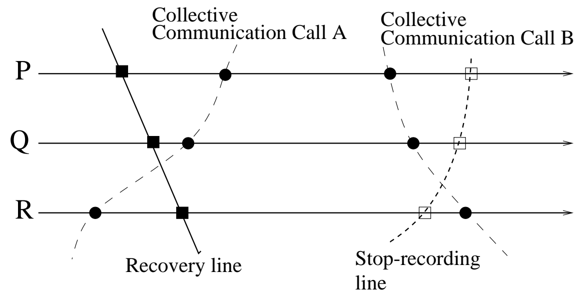


Fig. 4. Collective Communication

Most of the other collective communication calls can be handled in this way. Ironically, the only one that requires special treatment is `MPI_Barrier`, and the reason is that the MPI standard requires that no processor finishes a call to `MPI_Barrier` until every processor has started a call to `MPI_Barrier`.

Suppose that the collective communication call A in Figure 4 is an `MPI_Barrier`. Upon recovery, processors P and Q will have already finished their calls to `MPI_Barrier`, while R has not yet started its call. This is a clear violation of the required behavior. The solution is to ensure that all processes involved in a barrier execute it in the same epoch. In other words, barriers cannot be allowed to cross recovery lines.

A simple implementation is the following. All processes involved in the barrier execute an all-reduce communication just before the barrier to determine if they are all in the same epoch. If not, processes that have not yet taken their local checkpoints do so, ensuring that the barrier is executed by all processes in the same epoch. This solution requires the precompiler to insert the all-reduce communication and the potential checkpointing locations before each barrier. As shown in [3], the overhead of this addition is very small in practice.

7 State Saving

The protocols described in the previous sections assume that there is a mechanism for taking and restoring a local checkpoint on each processor, which we describe in this section.

Application State-Saving The state of the application running on each node consists of its position in the static text of the program, its position in the dynamic execution of the program, its local and global variables, and its heap-allocated structures. Our precompiler modifies the application source so that this state is correctly saved, and can be restarted, at the `potentialCheckpoint` positions in the original code. Our approach is similar to that used in the PORCH system[12]. While it currently only saves somewhat less data than system-level checkpointing, it offers two significant advantages over that approach. First, it is a starting point for optimizing the amount of state that is saved at a checkpoint. In Section 10, we describe ongoing work towards this

goal. Second, it is much simpler and more portable than system-level checkpointing, which very often requires modifying the operating system and native MPI library.

MPI Library State-Saving As was already mentioned, our protocol layer intercepts all calls that the application makes to the MPI library. Using this mechanism our system is able to record the direct state changes that the application makes (e.g., calls to `MPI_Attach_buffer`). In addition, some MPI functions take or return handles to opaque objects. The protocol layer introduces a level of indirection so that the application only sees handles to objects in the protocol layer (hereafter referred to *pseudo-handles*), which contain the actual handles to the MPI opaque objects. On recovery, the protocol layer reinitializes the pseudo-handles in such a way that they are functionally identical to their counterparts in the original process.

8 Performance

In this section, we present an overview of the full experimental results that can be found in [2] and [3].

We performed our experimental evaluation on the CMI cluster at the Cornell Velocity supercomputer. This cluster is composed of 64 2-way Pentium III 1GHz nodes, featuring 2GB of RAM and connected by a Giganet switch. The nodes have 40MB/sec bandwidth to local disk. The point-to-point experiments were conducted on 16 nodes, and the collective experiments were conductions on 32 nodes. On each node, we used only one of the processors.

8.1 Point-to-point

We evaluated the performance of the point-to-point protocol on three codes: a dense Conjugate Gradient code, a Laplace solver, and Neurosys, a neuron simulator. All the checkpoints in our experiments are written to the local disk, with a checkpoint interval of 30 seconds[2].

The performance of our protocol was measured by recording the runtimes of each of four versions of the above codes.

1. The unmodified program
2. Version #1 + code to piggyback data on messages
3. Version #2 + protocol's records and saving the MPI library state
4. Version #3 + saving the application state

Experimental results are shown in Figure 5.

We observe in the results that the overhead of using our system is small, except in a few instances. In dense CG, the overhead of saving the application state rises dramatically for the largest problem size. This is as a result of the large amount of state that must be written to disk. The other extreme is Neurosys, which has a very high communication to computation ration on the small problem size. In this case, the overhead of using the protocol becomes evident. For the larger problems it is less so.

[2] We chose such a small interval in order to amplify the overheads for the purposes of measurement. In practice, users would choose checkpoint intervals on the order of hours or days, depending upon the underlying system.

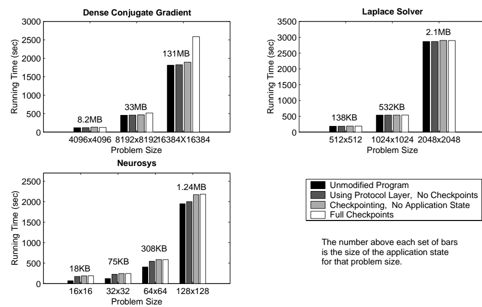


Fig. 5. Point-to-point Overheads

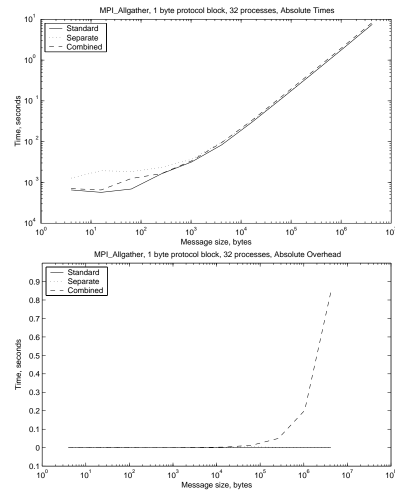


Fig. 6. `MPI_Allgather`

In our experiments, we initiated a new checkpointing 30 seconds after the last checkpoint was committed. For real applications on real machines, the developer will want to select a checkpoint frequency that carefully balances the overhead against the need to make progress. Since our protocol only incurs overhead *during the interval in which a checkpoint is being taken*, the developer can arbitrarily reduce the protocol overhead by reducing the frequency at which checkpoints are taken.

8.2 Collective

MPI supports a very large number of collective communication calls. Here, we compared the performance of the native version of `MPI_Allgather` with the performance of a version modified to utilize our protocol. Those modifications include sending the necessary protocol data (color and logging bits) and performing the protocol logic.

There are two natural ways to send the protocol data: either via a separate collective operation that precedes the data operation, or by "piggy-backing" the control data onto the message data and sending both with one operation. We have measured the overhead of both methods. The time for the separate operation case includes the time to send both messages. For the combined case, it includes the time to copy the control and message data to a contiguous region, to send the combined message, and to separate the message and protocol data on receipt.

The top graph in Figure 6 shows the absolute time taken by the native and protocol (both the separate and combined message) versions of `MPI_Allgather` for data message ranging in size from 4 bytes to 4 MB. The bottom graph shows the overhead, in seconds, that the two versions of the protocol add to the communication.

Examining the graphs, we see that for small messages, the relative overhead (percentage) might be high but the absolute overhead is small. For large messages sizes, the absolute overhead might be large, but relative to the cost of the native version, the cost is very small. This is the expected behavior. The next effect is that the observed overhead for real applications will be negligible.

9 Existing 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.

One such system is CoCheck [14], which provides fault-tolerance for MPI applications. CoCheck provides only the functionality for the coordination of distributed checkpoints, relying on the Condor [9] system 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.

Another distributed fault-tolerance implementation is the Manetho [5] system, which uses causal message logging to provide for system recovery. 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 many 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. 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 [13] system is another fault-tolerant system for MPI. Like CoCheck, it provides system-level checkpointing, and it has been implemented directly in the MPI layer. Like Manetho, it is primarily based upon message logging, and uses checkpointing to flush the logs when they grow too large.

10 Future Work

10.1 State savings

A 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 [1]. 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 technique we are developing is the detection of distributed redundant data. If multiple nodes each have a copy of the same data structure, only one of the nodes needs to include it in its checkpoint. On restart, the other nodes will obtain their copy from the one that saved it.

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.

10.2 Extending the protocols

In our current work, we are investigating the scalability of the protocol on large high-performance platforms with thousands of processors. We are also extending the protocol to other types of parallel systems. One API of particular interest is OpenMP [10], which is an API for shared-memory programming. Many high-performance platforms consist of clusters in which each node is a shared-memory symmetric multiprocessor. Applications programmers are using a combination of MPI and OpenMP to program such clusters, so we need to extend our protocol for this hybrid model.

On a different note, we plan to investigate the overheads of piggybacking control data on top of application messages. Such piggybacking techniques are very common in distributed protocols but the overheads associated with the piggybacking of data can be very complex, as our performance numbers demonstrate. Therefore, we believe that a detailed, cross-platform study of such overheads would be of great use for parallel and distributed protocol designers and implementors.

11 Conclusions

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 protocols for both point-to-point [2] and collective [3] operations to meet the need. These protocol can be used to provide fault tolerance for MPI programs without making any demands on or having knowledge of the underlying MPI implementation.

Used in conjunction with the method for automatically saving uniprocessor state described in [2], we have built a system that can be used to add fault-tolerance to C/MPI

programs. We have shown how the state of the underlying MPI library can be reconstructed by the implementation of our protocol. Experimental measurements show that the overhead introduced by the protocol implementation layer and program transformations is small.

Acknowledgments: This work was inspired by a sabbatical visit by Keshav Pingali to the IBM Blue Gene project. We would like to thank the IBM Corporation for its support, and Marc Snir, Pratap Pattnaik, Manish Gupta, K. Ekanadham, and Jose Moreira for many valuable discussions on fault-tolerance.

References

1. 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.
2. 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.
3. 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.
4. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.
5. 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.
6. 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.
7. R. Graham, S.-E. Choi, D. Daniel, N. Desai, R. Minnich, C. Rasmussen, D. Risinger, and M. Sukalski. A network-failure-tolerant message-passing system for tera-scale clusters. In *Proceedings of the International Conference on Supercomputing 2002*, 2002.
8. I. Gupta, T. Chandra, and G. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proc. 20th Annual ACM Symp. on Principles of Distributed Computing*, pages 170–179, 2001.
9. 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.
10. OpenMP. Overview of the OpenMP standard. Online at http://www.openmp.org/, 2003.
11. J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under UNIX. Technical Report UT-CS-94-242, Dept. of Computer Science, University of Tennessee, 1994.
12. B. Ramkumar and V. Strumpen. Portable checkpointing for heterogenous architectures. In *Symposium on Fault-Tolerant Computing*, pages 58–67, 1997.
13. S. Rao, L. Alvisi, and H. M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, Madison, Wisconsin, June 15 - 18, 1999.
14. G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.