Collective Operations in an Application-level Fault Tolerant MPI System

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

Abstract

The running times of many computational science programs are now significantly greater than the mean-time-between-failures (MTBF) of the hardware they run on. Therefore, fault-tolerance is becoming a critical issue on high-performance platforms.

Checkpointing is a technique for making programs fault tolerant by periodically saving their state and restoring this state after failure. In system-level checkpointing, the state of the entire machine is saved periodically on stable storage. This has too much overhead to be practical on high-performance platforms with thousands of processors. In practice, programmers do manual 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 an earlier paper, we presented a distributed checkpoint coordination protocol which handles MPI's point-to-point constructs, and deals with the unique challenges of application-level checkpointing. This protocol is implemented by 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. However, it did not handle collective communication, which is a very important part of MPI. In this paper we extend the protocol to handle MPI's collective communication constructs.

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, new trends in high-performance computing, such as the popularity of custom-assembled clusters, the dawn of grid computing, and increasing complexity of parallel machines, are increasing the probability of hardware failures, making it imperative that parallel programs tolerate hardware failures.

Unfortunately, many fault tolerance techniques developed by the distributed systems community do not scale well to parallel applications running on large parallel platforms such as the ASCI machines [10] which have thousands of processors. System-level checkpointing protocols [9] [12] require all processors to save 'core-dump' style snapshots of their computations periodically on stable storage; upon failure, all processors resume execution from the last snapshot. Unfortunately, the torrent of data saved at each checkpoint can overwhelm the disk storage system, so few high-performance machines support or encourage this style of obtaining fault-tolerance. Message logging approaches avoid writing to disk by logging messages in memory when they are sent; when a failed process is restarted, other processes help it to recover by replaying the messages they had sent it before it failed. Unfortunately, parallel programs communicate very frequently and send large amounts of data, so message logs can quickly fill up all the memory.

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, 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.

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

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 2.

In a previous paper [2], we argued that these problems can be circumvented by using a semi-automatic system for implementing application-level fault tolerance. With this system, the applications programmer inserts `PotentialCheckpoint()` calls at points in the program where it may be advantageous to take checkpoints. Our system does the rest. It consists of two parts, a *precompiler* and a *runtime co-ordination layer*.

1. The precompiler figures out what state needs to be saved at each potential checkpoint, and inserts code to save that state, and to restore it during recovery. Program analysis is used to reduce the state that is saved at each checkpoint. In this manner, we may gain the efficiency of manually inserted checkpointing, without the effort.

2. The co-ordination layer co-ordinates the checkpoints taken by different processes. It implements a novel protocol designed by us for non-blocking application-level checkpointing of MPI programs.

An overview of this system is given in Section 3.

Collective communication calls are an important part of the MPI standard [8]. One deficiency of our existing system is that it did not handle collective communication. In Section 4, we provide a taxonomy of collective communication constructs in MPI. We divide these calls into four groups depending on the directions of data flow in these calls. The co-ordination layer handles each of these groups differently. The protocol it implements for each group is described in Section 5. In Section 6, we provide experimental measurements of the overhead of the protocol for collective communication. We show that this overhead is acceptable. Finally, we conclude in Section 8 with a description of 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.

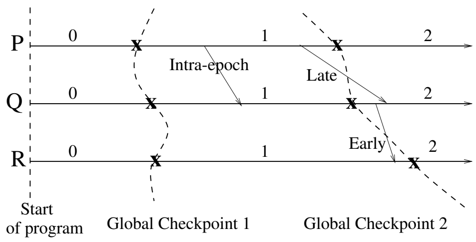


Figure 1: Epochs and message classification

2.1 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 all previous global checkpoints have 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.

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 1 shows examples of the three kinds of messages, using the execution trace of three processes named P, Q and R. 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.

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.

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 a program executes `PotentialCheckpoint` calls.

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, application-level checkpointing protocols must handle both late and early messages.

2.3 Handling late and early messages

We use Figure 1 to illustrate the issues associated with late and early messages.

Suppose that one of the processes in this figure fails after the taking of Global Checkpoint 2. 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 `recoveryLog` to save late messages after taking its local checkpoint; once logging is complete, the contents of this `recoveryLog` are saved on stable storage[1]. Late messages must be handled by many system-level checkpointing protocols as well.

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 `suppressList` to log early messages; once logging is complete, the `suppressList` is saved on stable storage.

Early messages also pose a separate and more subtle problem. The saved state of process R at Global Checkpoint 2 may depend on data contained in the early message from process Q. If that data were a random number generated by Q, R's state would be dependent on a non-deterministic event at Q. If this number is re-generated by Q on recovery, Q and R may disagree on its value after recovery.

In general, we must ensure that if a global checkpoint depends on a non-deterministic event, that event will reoccur the same way after restart. Therefore, mechanisms are needed to (i) log the non-deterministic events that a global checkpoint depends on, so that (ii) these events can be replayed during recovery.

[1]There is an entire class of fault-tolerance mechanics described in the distributed systems literature that is based upon the message logging. However, our use of message logging is fundamentally different than these. Whereas, the classic approaches logs all messages over the entire program execution, we log only subset of messages that occur within a small execution window. This difference is discussed further in Section 7.

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, than Q may require them in an order different from the order in which P sent them. It is important to note that this problem has nothing to do with the FIFO (or lack of) behavior of the underlying communication system; rather, it is a property of a particular application.

MPI also supports a very rich set of group communication calls called collective communication calls. These calls are used to do broadcasts, reductions, etc. 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 may need to be saved with checkpoints. It is not clear how this can be accomplished without access to the MPI library code. On the other hand, modifying the MPI library reduces the portability of our system.

3 The Point-to-point Protocol

In [2], we describe the coordination protocol for global checkpointing. This protocol handles point-to-point communication only. In this section, we will summarize this protocol and in Section 5, we extend this protocol to handle collective communication. The protocol is independent of the technique used by processes to take local checkpoints which are discussed in [2] and which we will not describe further here.

3.1 High-level description of protocol

Phase #1 To initiate a distributed snapshot, the initiator sends a control message called *pleaseCheckpoint* to all application processes. Each application process must take a local checkpoint at some time after it receives this request. In between, it sends and receives messages normally; otherwise, the program may deadlock.

Phase #2 When an application process reaches a point in the program where it can take a local checkpoint, it saves its local state and the identities of any early messages on stable storage. It then starts writing a log of (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[2], it sends a control message called *readyToStopLogging* back to the initiator, but continues to write non-deterministic decisions to the log.

Phase #3 When the initiator gets a *readyToStopLogging* message from all processes, it knows that every process has taken its local checkpoint. Since every process has transitioned to the new epoch, any message sent by any processor after the initiator has acquired this knowledge cannot be an early message. Therefore, all processes can stop logging. To share this information with the other processes, the initiator sends a control message called *stopLogging* to all other processes.

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

The second condition is a little subtle. Because we make no assumptions about message delivery order, it is possible for the following sequence of events to happen.

1. Process P receives a *stopLogging* message from the initiator, and stops logging.
2. P makes a non-deterministic decision.
3. P sends a message containing this decision to process Q, which is still logging.
4. Process Q uses this information to create an event that it logs.

When Q saves its log, we have a problem: 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 if it receives a message from a process that has itself stopped logging. These conditions for terminating logging can be described quite intuitively as follows: a process stops logging when it hears from the initiator or from another process that all processes have taken their checkpoints.

Once the process has saved its log on disk, it sends a *stoppedLogging* message back to the initiator. When the initiator receives a *stoppedLogging* message from all processes, it records on stable storage that the checkpoint that was just created is the one to be used for recovery, and terminates the protocol.

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

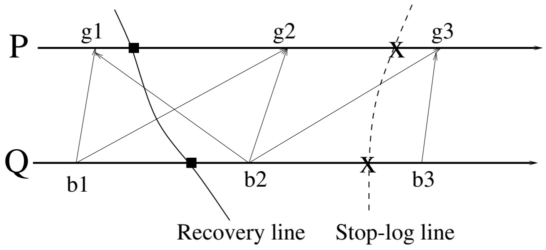


Figure 2: Possible Patterns of Communication

3.2 Guarantees provided by the protocol

It can be shown that this protocol provides certain guarantees that are useful for reasoning about correctness. First, we introduce the following terminology.

Definition 2 *In the context of a single global checkpoint, a process P is said to be*

- behind the recovery line *if it has not yet taken its local checkpoint,*
- beyond the recovery line *if it has taken its local checkpoint,*
- behind the stop-log line *if it is beyond the recovery line but has not stopped logging.*
- beyond the stop-log line *if it is beyond both the recovery line and the stop-log line.*

Claim 1 *The protocol described in this section provides the following guarantees.*

1. *No process can stop logging until all processes are beyond the recovery line.*
2. *A process P that is beyond the stop-log line cannot send a message to a process Q that is behind the stop-log line.*
3. *A process P that is beyond the stop-log line cannot receive a message from a process Q that is behind the recovery line.*

Figure 2 shows the possible communication patterns, given these guarantees. For example, a message sent by process Q at point b1 (behind the recovery line) cannot be received by process P at point g3 (beyond the stop-log line).

3.3 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 logging?

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 epoch in which the process is. It is initialized to 0 at start of execution, and incremented whenever that process takes a local checkpoint.
- *amLogging*: This is a boolean that is true when the process is logging, and false otherwise.
- *nextMessageID*: This is an integer which 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 logging. Finally, if the message is an early message, the receiver adds the pair <sender, messageID> to its `suppressList`. Each processor saves its `suppressList` to stable storage when it takes its local checkpoint. During recovery, each processor passes its list of messageID's to their sender processors so that resending these messages can be suppressed.

Further economy in piggybacking can be achieved if we exploit the fact that at most one global checkpoint can be ongoing at any time. This means that the epochs of processes can differ by at most one. Let us imagine that epochs are colored red and green alternatively. When the receiver is in a green epoch, and it receives a message from a sender in a green epoch, that message must be an intra-epoch message. If the message is from a sender in a red epoch, the message could be either a late message or an early message. It is easy to see that if the receiver is not logging, the message must be an early message; otherwise, it is a late message. Therefore, a process need only keep track of the color of its epoch, and this *color* bit can be piggybacked instead of the epoch number. With this optimization, the piggybacked information reduces to two booleans and an integer.

By exploiting the semantics of MPI, it is possible to eliminate the integer altogether, and piggyback only the two boolean values *color* and *amLogging*. We will not discuss this optimization further.

3.4 Completion of receipt 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. Protocols such as the Chandy-Lamport algorithm assume FIFO communication between processes, so they do not need explicit mechanisms to solve this problem. Since we cannot assume FIFO communication at the application level, we need to address this problem.

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.

A subtle issue is the following: 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 to assume that every process may communicate with every other process in every epoch, so a process expects to receive a $sendCount$ control message from every other process in the system. This solution works, but if the topology of the inter-process communication graphs is sparse, most $sendCount$ control messages will contain 0, which is wasteful. If the topology of this communication graph is sparse and fixed, we can set up a data structure in the protocol layer that holds this information. There are even fancier solutions for the case when the communication topology is sparse and dynamic, but we do not present them here.

3.5 Summary

A detailed description of the complete protocol for point-to-point communication can be found in [2]. The protocol requires each process to maintain the following variables; these are also used by the protocol for collective communication calls presented in Section 5.

- *color*: A single bit that denotes the color (red or green) of a processor. This bit is maintained by each processor and piggybacked onto each message to indicate the color of the sender.
- *amLogging*: A single bit indicating whether or not a processor is logging late messages and non-determinism. This bit is also maintained by each processor and piggybacked on each outgoing messages.
- *nextMessageID*: Each processor assigns each outgoing

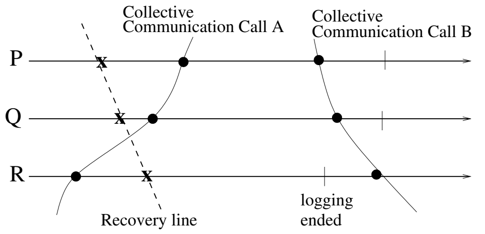


Figure 3: Collective Communication

message a unique ID. This ID is piggybacked on the message.

- *recoveryLog*: During the logging phase each processor maintains a log of all messages received and all non-deterministic events.
- *suppressList*: Each processor maintains a list of the early messages that it receives that must be suppressed upon recovery.

4 Classification of collective operations

The protocol described in the previous section must be extended to handle collective communication calls. The most obvious problem is that the processes participating in a collective communication call can straddle the recovery line in the sense that some of them might execute the call before taking their checkpoints while others might execute the call after taking their checkpoints, as shown in Call A in Figure 3. After recovery, process R will not re-execute the collective communication call but processes P and Q will, so the call will not complete correctly. A more subtle problem is illustrated by Call B in Figure 3. Suppose that R, which has stopped logging, broadcasts a value to process P that is still logging. If this value depends on a non-deterministic event at R that was not logged, and P logs this value, an inconsistent state may result after recovery.

To address such problems, it is convenient to divide MPI collective communication calls into four categories, based on the data flow of the communication.

1. *Single-sender*: One process sends data to the other processes in the communicator. Examples are `MPI_Bcast` and `MPI_Scatter`. The sending process is called the *root process* for that call, and data is said to flow from the root process to the other processes in the communicator.
2. *Single-receiver*: One process receives data from all the other processes in the communicator. Examples are `MPI_Gather` and `MPI_Reduce`. The receiving process is called the *root process* for that call, and data is said to flow to the root process from all other processes in the communicator.
3. *All-to-all communication*: Each process in the communicator sends and receives data to accomplish the collective communication. Examples are `MPI_Allgather` and `MPI_Alltoall`. All the processes are said to be root processes for that call, and data is said to flow from every process to every other process in the communicator.
4. *Barrier*: Unlike other communication calls, `MPI_Barrier` communicates no data since it is used to synchronize processes in a communicator.

The protocol developed in Section 5 provides the following guarantees which are similar to the guarantees for point-to-point communication of Claim 1.

Claim 2 *1. No process can stop logging until all processes are beyond the recovery line.*

2. In a collective communication call, data cannot flow from a process that is beyond the stop-log line to a process that is behind the stop-log line.

3. In a collective communication call, data cannot flow from a process P that is behind the recovery line to a process Q that is beyond the stop-log line.

Therefore, we see that if the arrows in Figure 2 are interpreted as directions of data flow, the figure shows the possible data flows for single-sender and single-receiver collective communication calls. For example, if process Q executes a broadcast at point *b1* (before taking its checkpoint), process P must receive this value before it crosses the stop-log line. Data flow in all-to-all communication is symmetric, so the possible data flows are simpler, and are shown in Figure 4. For example, a process P that has crossed the stop-log line cannot be involved in a collective communication call with a process Q that is beyond the recovery line but is still logging.

A word of caution is appropriate here. In most MPI implementations, collective calls are implemented using point-to-point communication. For example, broadcasts can be implemented in logarithmic time by using a fan-out tree of processes. It is important to distinguish the point-to-point messages that may be used in the underlying implementation of collective communication from the data flow directions in collective communication, such as the ones shown in Figure 4. Data flow directions are conceptual tools that we use to design the protocols discussed in Section 5; they are not necessarily related to the implementation of collective communication in the MPI library.

5 Protocol for handling collective operations

Our protocol treats each category of collective communication calls differently.

5.1 All-to-all collective operations

The protocol for all-to-all collective communication calls is relatively easy to understand, so we explain it first, using

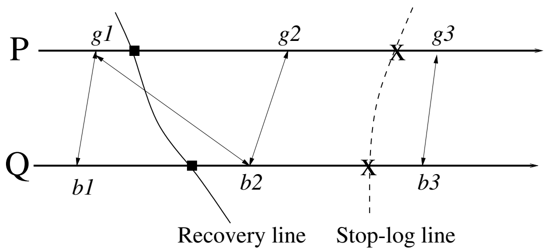


Figure 4: Data flow in all-to-all collective communication

MPI_Allreduce as an example. When the co-ordination layer intercepts an invocation of MPI_Allreduce, it executes the code shown in function our_MPI_Allreduce in Figure 5. This code should be understood with reference to Figure 4.

Suppose that the MPI_Allreduce straddles the recovery line (that is, there are at least two processes between which information flow is of the type $g1 \leftrightarrow b2$ in Figure 4). On recovery, processes that are behind the recovery line will not re-execute this call. Therefore, the protocol requires that if the MPI_Allreduce calls straddle the recovery line, processes that are beyond the recovery line (such as process Q at point $b2$) must log the result of the call, and replay this value on recovery.

The other possibility is that the MPI_Allreduce does not straddle the recovery line. If all processes are behind the recovery line (information flow is of the form $g1 \leftrightarrow b1$ in Figure 4), no process re-executes the call after recovery, so there is nothing to be done. If all processes are beyond the recovery line but behind the stop-log line (in Figure 4, information flow is of the form $g2 \leftrightarrow b2$), we require all processes to re-execute the call on recovery, so again there is nothing to be done. Otherwise, at least one of the processes has stopped logging. If so, this information is propagated to all other processes in the call which also stop logging (so information flow ends up being of the form $g3 \leftrightarrow b3$ in Figure 4). The result of the call is not logged, so all processes re-execute the call during recovery.

Putting all this together, we see that each process must send its color bit and its amLogging bit to other processes. By combining these bits as shown in Figure 5, all processes figure out whether the collective communication straddles the recovery line, and whether some process has stopped logging. This determination is implemented by two calls to MPI_Allreduce in the code in Figure 5. These two calls can be trivially combined into a single call; alternatively, the two bits can be piggybacked on the application data payload. We explore the relative overheads of these alternatives in Section 6.

```
our_MPI_Allreduce(send_data, recv_data, op, comm) {
    MPI_Allreduce(color, crosses_recovery_line, MPI_LXOR, comm);
    MPI_Allreduce(!amLogging, some_not_logging, MPI_LOR, comm);
    MPI_Allreduce(send_data, recv_data, op, comm);
    switch{
        case crosses_recovery_line && amLogging:
            recoveryLog.save(recv_data,comm);
        case !crosses_recovery_line && amLogging && some_not_logging:
            amLogging = false;
    }
}
```

Figure 5: Protocol for an all-to-all communication

5.2 Single-receiver collective operations

We use MPI_Reduce to illustrate how the protocol handles single-receiver collective communication calls. In Figure 2, process P is assumed to be the root for the collective communication call, and the arrows from process Q to process P show the information flows that can occur. When the root process of the collective communication invokes the call, it is either logging (in Figure 2, it is at point $g2$) or it is not logging (points $g1$ or $g3$ in Figure 2).

Suppose that the root process is behind the recovery line (point $g1$). If none of the other processes is beyond the recovery line, all information flow is of the form $b1 \rightarrow g1$. There is nothing to be done because no process executes the collective call during recovery. Otherwise, the collective call straddles the recovery line, and some of the information flows are of the form $b2 \rightarrow g1$. Since the root process will not execute the collective call during recovery, processes that execute the collective call after taking their checkpoints must suppress this call on recovery. The root process can identify such processes if each process sends the root its color, and note them in its suppressList. During recovery, these processes are informed that they must suppress these collective communication calls.

Suppose that the root process is beyond the recovery line and is logging (point $g2$). If the collective communication does not cross the recovery line, all processes execute the call during recovery and there is nothing to be done. If the collective communication crosses the recovery line (there is information flow of the form $b1 \rightarrow g2$), some of the processes will not invoke the collective communication call during recovery. Therefore, we require the root process to log the result of the call for replay during recovery; in addition, the root process identifies all processes that executed the call after taking their checkpoints, and notes them in its suppressList.

The final case is when the collective communication call does not cross the recovery line, and at least one of the senders has stopped logging. If so, the root process stops logging. An MPI_Reduce operation is used to inform the root whether any of the senders have stopped logging.

The pseudo-code in Figure 6 shows two collective communication calls for sending the `color` and `amLogging` information to the root. As always, these calls can be combined into one; the information can also be piggybacked on the application payload.

5.3 Single-sender collective operations

We use `MPI_Bcast` to illustrate how the protocol handles single-sender collective communication calls. In Figure 2, process Q is assumed to be the root for the collective communication call, and the arrows to process P show the data flows that can occur. When the root process of the collective communication invokes the call, it is either logging (in Figure 2, it is at point $b2$) or it is not logging (points $b1$ or $b3$ in Figure 2).

If the root process is behind the recovery line when it invokes the call (point $b1$), it does not re-execute the call on recovery. If the receiving process P performs the collective call while it is logging (point $g2$), its data flow straddles the recovery line, and process P must log the value it receives so it can replay this value on recovery. To enable P to discover if data flow crosses the recovery line, the root process Q must broadcast its color to the other processes.

Suppose that the root process is logging when it invokes the call (point $b2$). It is possible that one of the receiving processes has stopped logging (point $g3$); to enable it to recover, it is necessary for the root process to re-execute the broadcast during recovery. However, it is possible for one of the receiving processes to be behind the recovery line (point $g1$). Such a process would not participate in the collective call during recovery. To consume the message that it would be sent by the root process during recovery, the process logs the parameters of the call in a `reexecList`. On restart, calls in the `reexecList` are immediately invoked with the same parameters, except for the receive buffers, which are replaced with dummy arguments.

Finally, the root process may be beyond the stop-log line when it invokes the collective call. By broadcasting its amLogging bit to the other processes, it informs them that it has stopped logging, and they stop logging as well. Nothing needs to be logged because both the root process and the receivers re-execute the call during recovery.

In the code shown in Figure 7, the root process uses two calls to `MPI_Bcast` to broadcast its color and amLogging bits. As before, these two calls could be combined; the bits can also be piggy-backed on the application payload.

5.4 Barriers

The primary issue to consider with `MPI_Barrier` is the fact that it has explicit synchronization semantics. In other words, a process may not cross a barrier until all the other

```
our_MPI_Reduce(send_data, recv_data, op, root, comm)
{ if (my_proc_id != root_proc) {
        MPI_Gather(color, ..., root, comm);
        MPI_Reduce(!amLogging, ..., MPI_LOR, root, comm);
        MPI_Reduce(send_data,recv_data,op,root,comm);
    } else {
        /* I am receiving the data ... */
        MPI_Gather(..., colors, my_proc_id, comm);
        MPI_Reduce(..., some_not_logging, MPI_LOR,
                    my_proc_id, comm);
        MPI_Reduce(send_data,recv_data,op,my_proc_id,comm);

        bool crosses_recovery_line =
            exists { p || p != my_proc_id && colors[p] != color } ;
        switch {
            /* record early sends for suppression */
            case crosses_recovery_line && !amLogging:
                { foreach (p in comm where i != my_proc_id) {
                    if (colors[p] != color)
                    suppressList.save(comm,p);
                }
            }
            /* log received data */
            case crosses_recovery_line && amLogging:
                { recoveryLog.save(recv_data, comm);
                /* record all sends that do _not_ cross recovery
                    line for suppression */
                foreach (p in comm where i != my_proc_id) {
                    if (colors[p] == color)
                    suppressList.save(comm,p);
                }
            }
            case !crosses_recovery_line && amLogging &&
                some_not_logging:
                { amLogging = false;
                break;
                }
            }
        }
    }
}
```

Figure 6: Protocol for single-receiver collective communication

```
our_MPI_Bcast(data, root_proc, comm)
{
    if ( my_proc_id == root_proc ) {
        /* I am sending the data ... */
        MPI_Bcast(color, my_proc_id, comm);
        MPI_Bcast(amLogging, my_proc_id, comm);
        MPI_Bcast(data, root_proc, comm);
    } else {
        /* I am receiving the data ... */
        MPI_Bcast(root_color, root_proc, comm);
        MPI_Bcast(root_is_logging, root_proc, comm);
        MPI_Bcast(data, root_proc, comm);

        bool crosses_recovery_line = (color != root_color);

        switch {
            /* log late Bcast */
            case crosses_recovery_line && amLogging:
                { recoveryLog.save(recv_data, comm);
                break;
                }
            /* will have to reexec early Bcast */
            case crosses_recovery_line && !amLogging:
                { reexecList.save(comm);
                break;
                }
            /* turn off logging */
            case !crosses_recovery_line && amLogging &&
                !root_is_logging:
                { amLogging = false;
                break;
                }
        }
    }
}
```

Figure 7: Protocol for single-sender collective communication

```
our_MPI_Barrier(comm)
{
    /* exchange information about each processor's current state */
    MPI_Allgather(color, colors, comm);
    MPI_Reduce(!amLogging, some_not_logging, MPI_LOR, comm);

    bool somebody_checkpointed =
        exists { p — p != my_proc_id && colors[p] != color } ;

    switch {
        /* if we haven't checkpointed but somebody else has */
        case !amLogging && somebody_checkpointed:
            {Take a Checkpoint Immediately
            break;
            }
        /* if we're logging but somebody has already stopped */
        case amLogging && some_not_logging:
            { amLogging = false;
            break;
            }
    }

    /* finally, perform the actual barrier */
    MPI_Barrier(comm);
}
```

Figure 8: Protocol for barriers

processes have reached the barrier. Consider what happens when a call to MPI_Barrier crosses the recovery line. We have process P that calls MPI_Barrier ahead of the recovery line and process Q that calls MPI_Barrier behind the recovery line. On recovery Q will recover at a point past the barrier while P will recover in a state before it reached the barrier. Clearly, the very fact that the barrier crosses the recovery line violates MPI_Barrier's synchronization semantics.

The solution to this problem is to ensure that barriers may never cross recovery lines, an invariant that can be enforced by placing a special BarrierPotentialCheckpoint location before each call to MPI_Barrier. At this BarrierPotentialCheckpoint location each process will check whether any other other process has taken a checkpoint. If so then this process will also take a checkpoint. As a result we can be sure that if any process takes a checkpoint before an MPI_Barrier, the entire recovery line will be behind that MPI_Barrier on all the processors participating in the call.

In the code shown in Figure 8 the our_MPI_Barrier function contains both the code for taking a checkpoint and performing a barrier. The calls to MPI_AllGather and MPI_AllReduce exchange the control information that helps us determine both whether to take a checkpoint and whether to stop the log. As usual, these two calls can be combined and/or piggybacked.

6 Experiments

In this section, we report on experiments that were performed to measure the overhead added by this protocol to MPI's native collective communication operations.

Our experiments were conducted on a 32 node computational cluster, part of the Velocity Cluster at the Cornell Theory Center [1]. Each node contains 2 Intel Pentium III processors, each running at 1.0 GHz, and has 2 GB of RAM. The nodes are connected by the Giganet cLAN [7] interconnect. The operating system is Windows 2000. The MPI implementation is MPI/Pro 1.6.4 using the Virtual Interface Architecture (VIA) [3]. For our experiments, we only ran one MPI process (rank) on each node.

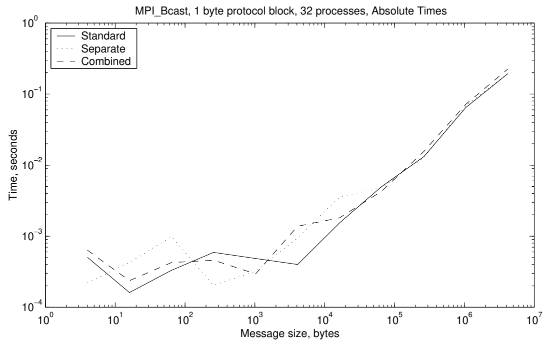
MPI supports a very large number of collective communication calls. From these, we selected `MPI_Bcast`, `MPI_Gather`, `MPI_Allgather`, and `MPI_Barrier` for our experiments since each of them represents one of the classes of MPI calls discussed in Section 4. We compared the performance of the native version of that operation 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 shown in the pseudo-code in Section 5. The color and logging bits were sent together as a one byte block.

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. For comparison purposes, we implemented both methods. The overhead for the separate operation case includes the time to send both messages. For the combined case, it includes the time to copy the message data and the control data to a contiguous region, the time for the single communication, and the time to separate the message and protocol data on receipt.

The top graph in Figure 9 shows the absolute time taken by the native and protocol (both the separate and combined message) versions of `MPI_Bcast` for data message ranging in size from 4 bytes to 4 MB. Both axes of this graph have logarithmic scales. The bottom graph shows the overhead, in seconds, that the two versions of the protocol add to the communication. Figure 10 shows similar information for `MPI_Gather` and Figure 11 does so for `MPI_Allgather`.

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.

Examining the second graph in each set, we observe that the cost of using the separate message protocol is fairly constant, whereas the cost of the "piggy-backed" protocol grows with the size of the message. These behaviors are to be expected: using a separate message imposes a fixed cost, regardless of the size of the data message, while using a combined message requires copying at both the sender(s) and the receiver(s). Therefore, the optimal strategy would be a protocol that switched from a combined message to separate messages as the size of the data message grew. Using such a strategy, the overhead added by this protocol is minimal.



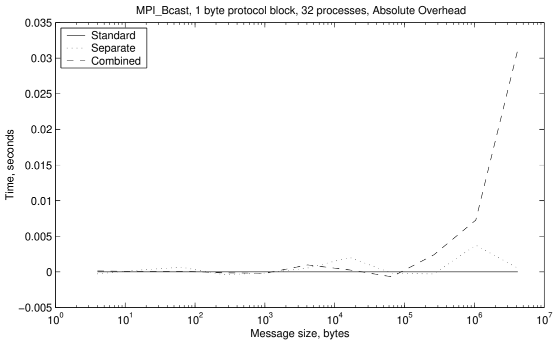


Figure 9: `MPI_Bcast`

Although a collective communication, `MPI_Barrier` does not actually communicate any message data to the application processes. Therefore, we do not have the option of piggy-backing the protocol data, and we must use must use a separate communication to send it. Additionally, experimental results for barrier do not depend on message size. Figure 12 compares the relative performance of the native `MPI_Barrier` and the protocol version. Our experiments only compare the communication costs of the native and the protocol versions of `MPI_Barrier`. The difference in the communication times of the two versions is seen to be inconsequential. Note that the true cost of a barrier is the cost of waiting for all the processes to arrive at it: this cost is usually much greater than the cost of actual communication that the barrier requires. Therefore, we conclude that the overhead of our protocol is negligible.

7 Prior Work

While much theoretical work has been done in the field of distributed fault tolerance, there exist few systems that im-

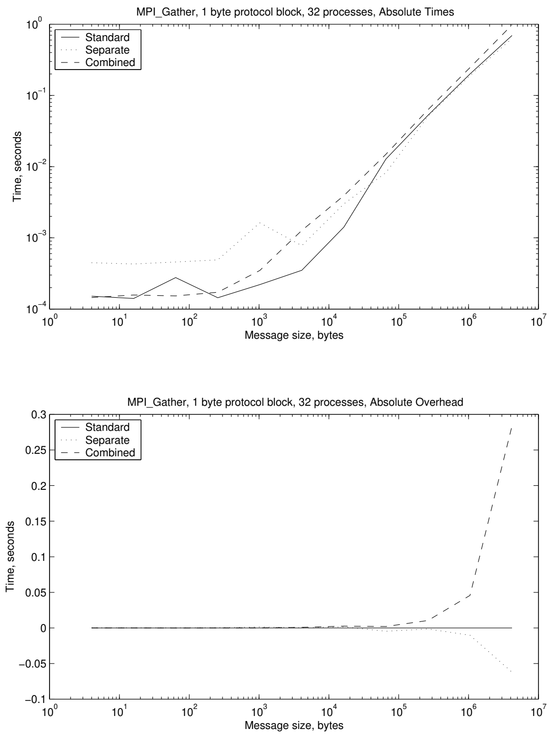


Figure 10: `MPI_Gather`

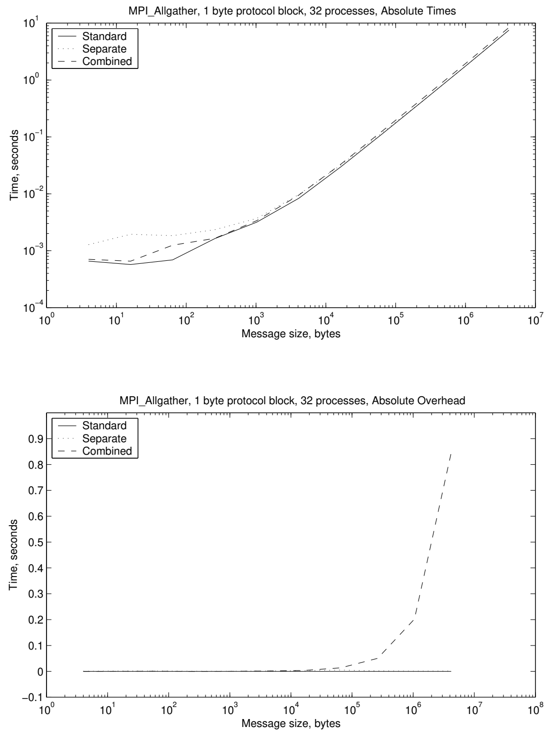


Figure 11: `MPI_Allgather`

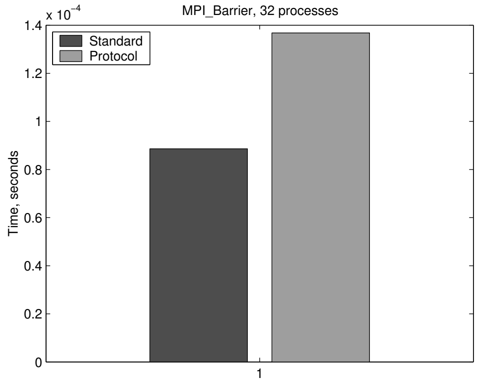


Figure 12: `MPI_Barrier`

plement it 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. The key difference between CoCheck and our work is that whereas our protocol is independent of the underlying implementation of MPI, CoCheck is integrated with its own MPI implementation. CoCheck assumes that collective communications are implemented as point-to-point messages, an assumption they can make because they provide their own MPI implementation, but that can not be made when attempting to provide fault tolerance for high-performance implementations of the standard.

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 it requires a process to take a checkpoint whenever these logs get too large, it is not applicable to application-level checkpointing.

Although our protocol also requires the use of a log, our log is used only while checkpointing to ensure that the checkpoints are consistent.

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 has been implemented directly in the MPI layer and it provides system-level checkpointing.

Like Manetho, it is primarily based upon message logging, and uses checkpointing to flush the logs when they grow too large.

8 Conclusions and Future Work

In this paper we have presented a distributed checkpointing protocol capable of handling the unique requirements of application-level checkpointing. In conjunction with a single-processor checkpointer (like the one described in [2]) this protocol can be used to provide fault tolerance for MPI programs without making any demands on or having knowledge of the underlying MPI implementation.

We have presented the protocol's overall structure, including the semantics of its log. We then showed how to use the guarantees provided by the basic protocol to checkpoint MPI point-to-point messages and collective communications. Finally. we measured the overheads associated with the additional data our protocol attaches to communications and the additional code executed for each communication call and discovered that these overheads were low.

Having shown that it is possible to checkpoint MPI at the application-level using no knowledge of the underlying implementation, we would like to extend out work to other types of parallel systems. One API of particular interest is the OpenMP [11] shared memory interface standard. Of primary use on Shared Memory Multiprocessors, OpenMP presents a release-consistency-based API which has a number of complex features such as locks, parallel for loops and barriers.

Another interesting extension of our current work is the systematic evaluation of the overheads of piggybacking control data on top of network communications. Such piggybacking techniques are very common in distributed protocols but as the performance numbers that we collected for our own protocol indicate, the overheads associated with the piggybacking of data can be very complex. Therefore we believe that a detailed, cross-platform study of such overheads on top of MPI implementations is in order and would be of great use for parallel and distributed protocol designers and implementors.

References

[1] Cornell theory center. Online at http://www.tc.cornell.edu/, 2003.

[2] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs, 2002. submitted to the Symposium on Principles and Practice of Parallel Programming.

[3] D. Cameron and G. Regnier. *The Virtual Interface Architecture*. Intel Press, San Francisco, California, first edition, 2002.

[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] Emulex corporation. Overview of giganet clan. Online at http://www.emulex.com/ts/legacy/clan/index.htm, 2003.

[8] M. P. I. Forum. Overview of the mpi standard. Online at http://www.mpi-forum.org/, 2003.

[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] National Nuclear Security Administration. Asci home. Online at http://www.nnsa.doe.gov/asc/, 2002.

[11] OpenMP. Overview of the openmp standard. Online at http://www.openmp.org/, 2003.

[12] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under unix. Technical Report UT-CS-94-242, 1994.

[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.