

Collective Operations in an Application-level Fault Tolerant System

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

Abstract

1 Introduction

The problem of implementing software systems that can tolerate hardware failures has been studied extensively by the distributed systems community. 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 with thousands of processors, such as the ASCI machines. System-level checkpointing protocols 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 compu-

tational 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 amino-acids 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 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, 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 is required only to specify points in the program where it may be advantageous to take checkpoints, by inserting `PotentialCheckpoint()` calls in such places. 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 minimize the state that is saved at each checkpoint. In this manner, we 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 co-ordination protocol designed by us for non-blocking

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

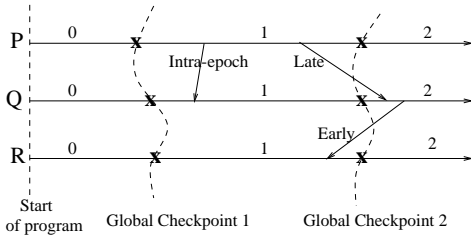


Figure 1: Epochs and message classification

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. 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 7 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 the existing protocols for non-blocking parallel checkpointing, which were designed for system-level checkpointers, are not suitable when the state saving occurs at the application level. In Section 3, we show how these difficulties are overcome with our approach.

2.1 Terminology

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 global 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 which Q sent after taking 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. On restart, each processes will resume execution from its state as saved in the checkpoint. For process Q to recover correctly, it must obtain the late message that was sent to it by process P prior to the failure. However, process P will not resend this message because the send occurred before P took its checkpoint. 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 `recovery_log` to save late messages after taking its local checkpoint; once logging is complete, the contents of this log are saved on stable storage. Late messages must be handled by system-level checkpointing protocols as well.

Early messages, such as the message sent from process Q to process R pose a different problem. Process R received this message before taking its checkpoint; after recovery it does not expect to be resent this message. For the application to be correct, therefore, process Q must suppress resending this message. 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 `suppress_log` to log early messages; once logging is complete, the `suppress_log` 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 the data contained in the early message from process Q . If that data was a random number generated by Q , R 's state would be dependent on a non-deterministic event at Q . If the number was generated after Q took its checkpoint, then on restart, Q and R may disagree on its value.

In general, we must ensure that if a global checkpoint depends on a non-deterministic event, that event will re-occur 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.

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, a process Q may use tags to receive messages from another process P 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 A Non-Blocking, Coordinated Protocol for Application-level Checkpointing

We now describe the coordination protocol for global checkpointing. This protocol handles point-to-point communication only. 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 we will not describe further in this paper.

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, but it is free to send and receive as many messages as it likes between the time it is asked to take a checkpoint and when it actually complies with this request.

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 messages¹, 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.

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

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.

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,*
- beyond the stop-log line *if it is beyond the recovery line and has stopped logging, and*
- behind the stop-log line *if it is not beyond 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.*

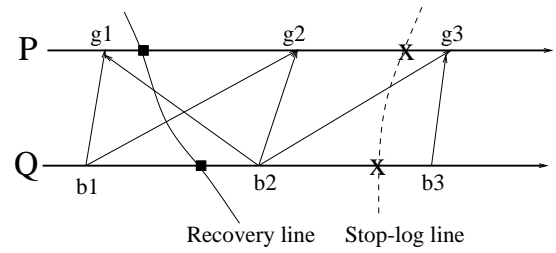


Figure 2: Possible Patterns of Communication

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 logs the pair <sender, messageID>. These pairs

are saved to stable storage when the processor takes its local checkpoint. During recovery, these pairs are retrieved from stable storage by the receivers of these messages, and the senders of these early messages are informed of the messageIDs 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.

It can be shown that it is sufficient to piggyback only the two boolean values *color* and *amLogging*.

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 minor detail is that a process P actually needs to keep *two* receive counts for each process Q that may send it messages; this is because late messages from P to Q sent in one epoch may be interspersed with intra-epoch messages from P to Q sent in the next epoch. In the protocol given below, these two counters are called *previousReceiveCount* and *currentReceiveCount*.

A more 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. In the pseudo-code of Figure 3, we assume that the inter-process communication graph is fixed, and we use the terms *senders* and *receivers* to denote the set of processes that send messages to a given process, and the set of processes that are sent messages by a given process respectively.

3.5 Putting it all together

Figure 3 is a synthesis of the mechanisms discussed above into a single protocol which is executed by the protocol layer at each processor, p .

Each process maintains the following variables:

- *epoch*: The current epoch number. Initialized to 0.
- *amLogging*: whether or not logging of late messages and non-determinism is occurring. Initialized to false.
- *nextMessageID*: The ID of the next message sent. Initialized to 0.
- *checkpointRequested*: True if a local checkpoint should be taken at the next call to `potentialCheckpoint`. Initialized to false.
- *sendCount*[q]: Number of messages sent to processor q during the current epoch. Initialized to 0.
- *earlyIDs*[q]: ID's of early messages received from processor q . Initialized to nil.
- *currentReceiveCount*[q]: Number of intra-epoch messages received from processor q . Initialized to 0.
- *previousReceiveCount*[q]: Number of late messages received from processor q . Initialized to 0.
- *totalSent*[q]: Number of messages sent by processor q before it took its last checkpoint. Initialized to ∞ .

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 Fig-

```

communicationEventHandler()
  Application message send to process  $d$ :
    Piggyback  $\langle epoch, amLogging, nextMessageID \rangle$ 
      on the message
     $sendCount[d]++$ 
     $nextMessageID++$ 
  Application message receive from process  $u$ :
    Remove  $\langle epoch_u, amLogging_u, messageID_u \rangle$ 
      from the message
    early message://assert not  $amLogging$ 
      append  $messageID_u$  to  $earlyIDs[u]$ 
    intra-epoch message:
      if ( $amLogging$  and not  $amLogging_u$ )
        finalizeLog()
         $currentReceiveCount[u]++$ 
    late message://assert  $amLogging$ 
      append message to log
       $previousReceiveCount[u]++$ 
      receivedAll?()

Control message:  $pleaseCheckpoint$ 
   $checkpointRequested \leftarrow true$ 
Control message:  $stopLogging$ 
  finalizeLog()
Control message:  $mySendCount(n)$  from process  $u$ 
   $totalSent[u] \leftarrow n$ 
  if ( $amLogging$ )/ $p$  has taken its own checkpoint
    receivedAll?()

receivedAll?()
  if (for all senders  $u$ 
     $previousReceiveCount[u] = totalSent[u]$ )
    send  $readyToStopLogging$  message to initiator
     $totalSent[u] \leftarrow \infty$  for all senders  $u$ 

finalizeLog()
  write log to stable storage
   $amLogging \leftarrow false$ 
  send  $StoppedLogging$  message to initiator

potentialCheckpoint()
  if ( $checkpointRequested = false$ ) return
  save node state to stable storage (see Section ??)
   $epoch++$ 
  for each receiver  $d$ 
    send  $mySentCount(sendCount[d])$  to  $d$ 
  for each sender  $u$ 
     $previousReceiveCount[u] = currentReceiveCount[u]$ 
     $currentReceiveCount[u] = length(earlyIDs[u])$ 
    save  $earlyIDs[u]$  to stable storage
     $earlyIDs[u] \leftarrow nil$ 
   $checkpointRequested \leftarrow false$ 
   $amLogging \leftarrow true$ 
   $nextMessageID \leftarrow 0$ 
  receivedAll?()

```

Figure 3: Application-level Checkpointing Protocol

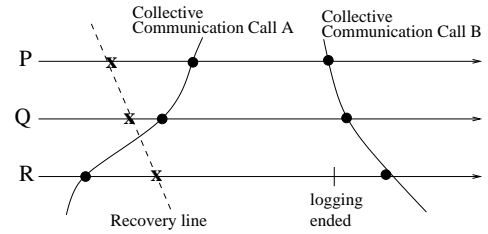


Figure 4: Collective Communication

Figure 4. 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 4. 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* 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

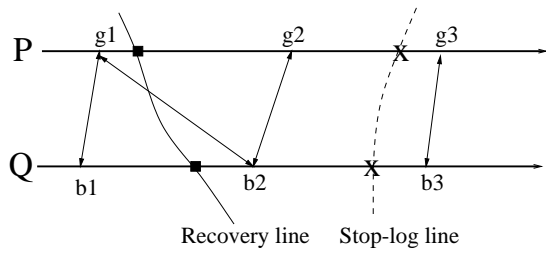


Figure 5: Data flow in all-to-all collective communication

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 5. 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 5. Data flow directions are conceptual tools that we use to design the protocols discussed in Section 5; they are not necessarily related in any way to how collective communication is actually implemented.

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 `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 6. This code should be understood with reference to Figure 5.

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 $g1$ - $b2$ in Figure 5). 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.

Alternatively, the `MPI_Allreduce` does not straddle the recovery line. If all processes are behind the recovery line (information flow is of the form $g1$ - $b1$ in Figure 5), 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 5, information flow is of the form $g2$ - $b2$), 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 $g3$ - $b3$ in Figure 5). 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 appropriately, all processes figure out whether the collective communication straddles the recovery line, and whether some process has stopped logging. This is implemented by two calls to `MPI_Allreduce` in the code in Figure 6. These two calls can be trivially combined into a single call; alternatively, the two bits can be piggy-backed on the application data payload. We explore the relative overheads of these alternatives in Section 6.

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 are beyond the recovery line, all information flow is of the form $b1$ - $g1$. 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 $b2$ - $g1$. 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 `suppress_log`. During recovery, these processes are informed that they must suppress these collective

```

our_MPI_Allreduce(send_data, recv_data, op, comm)
{
    if (recovery_log.matches(comm)) {
        recv_data = recovery_log.consume(comm);
        return;
    }
    MPI_Allreduce(my_color, crosses_recovery_line, MPI_LXOR, comm);
    MPI_Allreduce(!i_am_logging, some_not_logging, MPI_LOR, comm);
    MPI_Allreduce(send_data, recv_data, op, comm);
    switch{
    case crosses_recovery_line & amLogging:
        recovery_log.save(recv_data, comm);
    case !crosses_recovery_line & amLogging & some_not_logging:
        amLogging = false;
    }
}

```

Figure 6: Protocol for an all-to-all communication

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 process 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-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 `suppress_log`.

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 7 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 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 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 information 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 its information 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 call in a `re-exec log`; on restart, calls in the `re-exec log` are invoked 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 8, 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 could even be piggy-backed on the application payload.

6 Experiments

Greg writes this after he gets numbers.

7 Conclusions and Future Work

```

our_MPI_Reduce(send_data, recv_data, op, root, comm)
{
    if (my_proc_id != root_proc) {
        /* I am sending the data ... */
        if (suppress_log.matches(comm)) {
            suppress_log.consume(comm);
            return;
        }
        MPI_Gather(my_color, ..., root, comm);
        MPI_Reduce(!i_am_logging, ..., MPI_LOR, root, comm);
        MPI_Reduce(send_data,recv_data,op,root,comm);
    } else {
        /* I am receiving the data ... */
        if (recovery_log.matches(comm)) {
            data = recovery_log.consume(comm);
            return;
        }
        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] != my_color};
    switch {
        case crosses_recovery_line & !amLogging: /* record early sends for suppression */
            {foreach (p in comm where i != my_proc_id) {
                if (colors[p] != my_color)
                    new_suppress_log.save(comm,p);
            }
        case crosses_recovery_line & amLogging: /* log received data */
            {log.save(recv_data, comm);
            /* record all sends that do _not_ cross recovery for suppression */
            foreach (p in comm where i != my_proc_id) {
                if (colors[p] == my_color)
                    new_suppress_log.save(comm,p);
            }
        case !crosses_recovery_line & amLogging & some_not_logging:
            {amLogging = false;
            break;
            }
        }
    }
}
}

```

Figure 7: 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(my_color, my_proc_id, comm);
        MPI_Bcast(i_am_logging, my_proc_id, comm);
        MPI_Bcast(data, root_proc, comm);
    } else {
        /* I am receiving the data ... */

        if (recovery_log.matches(comm)) {
            data = recovery_log.consume(comm);
            return;
        }

        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 = (my_color != root_color);

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

```

Figure 8: Protocol for single-sender collective communication