Automated Application-level Checkpointing of MPI Programs

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

Abstract

Because of increasing hardware and software complexity, the running time of many computational science applications is now more than the mean-time-to-failure of high-peformance computing platforms. Therefore, computational science applications need to tolerate hardware failures.

In this paper, we focus on the stopping failure model in which a faulty process hangs and stops responding to the rest of the system. We argue that tolerating such faults is best done by an approach called application-level coordinated non-blocking checkpointing, and that existing fault-tolerance protocols in the literature are not suitable for implementing this approach.

In this paper, we present a suitable protocol, and show how it can be used with a precompiler that instruments C/MPI programs to save application and MPI library state. An advantage of our approach is that it is independent of the MPI implementation. We present experimental results that argue that the overhead of using our system can be small.

1 Introduction

Fault-tolerant programming has been studied extensively in the context of distributed systems [6]. In contrast, the high-performance parallel computing community has not devoted much attention to this problem because hardware failures in parallel platforms were not frequent enough to be a cause for concern. Most high-performance computing was done on "big-iron platforms": monolithic vector or parallel computers that were designed, built, and maintained by a single vendor. Because these machines cost many millions of dollars, vendors could afford to design reliable components and integrate them carefully to produce relatively robust computing platforms. Moreover, unlike distributed systems programs such as air-traffic control systems that must run without stopping, most computational science programs ran for durations that were much less than the mean-time-between-failure (MTBF) of the underlying hardware.

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

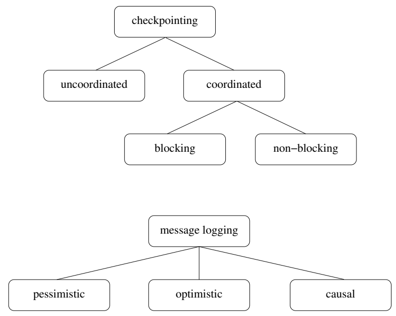

Figure 1: Hierarchy of different fault tolerance techniques

Recent changes in the high-performance parallel computing world are bringing the issue of fault-tolerance to the front and center. First, the number of processors in big-iron machines is increasing rapidly: the recently announced Blue Gene/L will have over 130,000[18]. Anecdotal evidence is that such a machine loses a processor every few hours; increasing the number of processors increases the overall performance, but it also increases the number of points of failure. Second, parallel computing is shifting from expensive monolithic hardware systems to low-cost, custom-assembled clusters of processors and communication fabric. The recent trend towards Internet-wide grid-computing is another change in the hardware picture that increases the probability of hardware failures during program execution. Third, many computational science programs are now designed to run for days or even months at a time; some examples are the ASCI stockpile certification programs[13] and *ab initio* protein-folding programs such as IBM's Blue Gene [9] codes which are intended to run for months.

Therefore, the running times of many applications are now significantly longer than the MTBF of the underlying hardware. Computational science programs must tolerate hardware failures.

1.1 Problem Definition

To address this problem, it is necessary to define the fault model. Two common classes of models are *Stopping* and

Byzantine [11]. In a Stopping model, a faulty process hangs and stops responding to the rest of the system, neither sending nor receiving messages. Byzantine faults permit a faulty process to perform more damaging acts such as sending corrupted data to other processes.

In this paper, we focus our attention on stopping processes. As we discuss in this paper, there are many interesting problems to be solved even in this restricted domain. Moreover, a good solution for this failure model can be a useful mechanism in addressing the more general problem of Byzantine faults.

In general, good abstractions are key to effective handling of failures. In this spirit, we make the standard assumption that there is a reliable transport layer for delivering application messages, and we build our solutions on top of that abstraction. One such reliable implementation of the MPI communication library is Los Alamos MPI (LA-MPI) [7].

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

1.2 Solution space

Figure 1 classifies some of the ways in which programs can be made fault-tolerant. An excellent survey of these techniques can be found in [6].

Checkpointing techniques periodically save a description of the state of a computation to stable storage; if any process fails, all processes are rolled back to the last checkpoint, and the computation is restarted from there. *Message-logging* techniques in contrast require restarting only the computation performed by the failed process. Surviving processes are not rolled back but must help the restarted process by replaying messages that were sent to it before it failed. The simplest implementation of message logging requires every process to save a copy of every message it sends. A more sophisticated approach might try to regenerate messages on demand using approaches like reversible computation. Although message-logging is a very appealing idea which has been studied intensively by the distributed systems community [5, 10, 16], our experience is that the overhead of saving or regenerating messages tends to be so overwhelming that the technique is not competitive in practice. This may be because parallel programs communicate more data more frequently than distributed programs [17].

We therefore focus on checkpointing.

Checkpointing techniques can be classified along two independent dimensions.

(1) The first dimension is the abstraction level at which the state of a process is saved. In *system-level checkpointing*, the bits that constitute the state of the processm such as the contents of the program counter, registers and memory, are saved on stable storage. Examples of systems that do system-level checkpointing are Condor[12] and Libckpt[14]. Some systems like Starfish[1] give the programmer some control on what is saved. Unfortunately, complete system-level checkpointing of parallel machines with thousands of processors can be impractical because each system checkpoint can require thousands of nodes sending terabytes of data to stable storage. For this reason, system-level checkpointing is not done on large machines such as the IBM Blue Gene or the ASCI machines.

One alternative which is popular is *application-level checkpointing*. Applications can obtain fault-tolerance by providing their own checkpointing code[3]. The application is written such that it correctly restarts from various positions in the code by storing certain information to a restart file. The benefit of this technique is that that the programmer needs only save the minimum amount of data necessary to recover the program state. For example, in an *ab initio* protein folding code, it suffices to save the positions and velocities of the various bases, which is a small fraction of the total state of the parallel system. The disadvantage of this approach to implementing application-level checkpointing is that it complicates the coding of the application program, and it is one more chore for the parallel programmer.

In this paper, we explore the use of compiler technology to automate application-level checkpointing.

(2) The second dimension along which checkpointing techniques can be classified is the technique used to coordinate parallel processes when checkpoints need to be taken. In uncoordinated checkpointing, each process saves its state whenever it wants to without coordinating with other processes. Although this is simple, restart can be problematic due to exponential rollback, which may cause the computation to roll so far back that is makes no progress [6]. For this reason, uncoordinated checkpointing has fallen out of favor.

Coordinated checkpointing can be divided into blocking and non-blocking checkpointing. Blocking techniques bring all processes to a stop before taking a global checkpoint. Hardware blocking was used on the IBM SP-2 to take system-level checkpoints. Software blocking techniques exploit barriers - when processes reach a global barrier, each one saves its own state on stable storage. This is essentially the solution used today by applications programmers who roll their own application-level state-saving code. However, this solution can fail for some MPI programs since MPI allows messages to cross barriers. These messages would not be saved with the global checkpoint. Moreover, new data-driven programming styles are eschewing the global barriers, ubiquitous in BSP-style bulk-synchronous programs, in favor of fine-grain, data-oriented synchronization. Such programs may not have barriers, and there may be no safe places in the code in which barriers can be inserted without creating

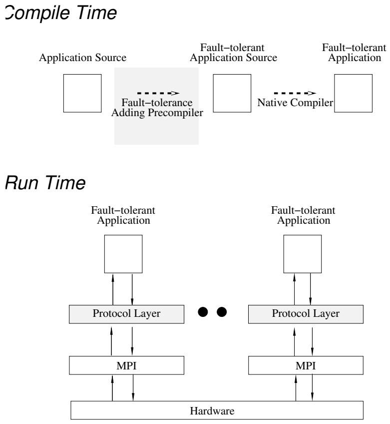

Figure 2: System Architecture

deadlocks.

For these reasons, *non-blocking* coordinated checkpointing is an interesting alternative. In this approach, a global coordination protocol, implemented by exchanging special marker or control tokens, is used to orchestrate the saving of the states of individual processes and the contents of certain messages, to provide a global snapshot of the computation from which the computation can be restarted. A distinguished process called the initiator is responsible for initiating and monitoring the protocol; to take a local checkpoint, an application process may communicate with other application processes but it makes no assumptions about the states of other processes. The Chandy-Lamport protocol is perhaps the most well-known non-blocking protocol [4]. Unfortunately, these protocols were designed to work with system-level checkpointing — as we discuss in Section 3, there are fundamental difficulties in using them for application-level checkpointing.

Therefore, we have developed a new protocol for non-blocking coordination that works smoothly with application-level state-saving.

1.3 Overview of our approach

In this paper, we discuss the use of compiler technology to implement application-level, coordinated, non-blocking checkpointing of MPI programs.

Figure 2 is an overview of our approach. The CCIFT (Cornell Compiler for Inserting Fault-Tolerance) *precompiler* reads almost unmodified single-threaded C/MPI source files and instruments them to perform application-level state-saving; the only additional requirement for the programmer is that he insert calls to a function called `PotentialCheckpoint` at points in the application where the programmer wants checkpointing to occur. We have not yet implemented optimizations to reduce the amount of state that is saved, so the instrumented code saves the entire state when it takes a checkpoint. The output of this precompiler is compiled with the native compiler on the hardware platform, and is linked with a library that constitutes a *protocol layer* for implementing the non-blocking coordination. This layer sits between the application and the MPI layer, and intercepts all calls from the instrumented application program to the MPI library[1]

This design permits us to implement the coordination protocol without modifying the underlying MPI library, which promotes modularity and eliminates the need for access to MPI library code which is proprietary on some systems. Further, it allows us to easily migrate from one MPI implementation to another.

The rest of this paper is organized as follows. We introduce some notation and terminology in Section 2. In Section 3, we discuss the main hurdles that must be overcome to implement our solution, and argue that the coordination protocols in the literature cannot be used for our problem. In Section 4, we present our solutions to these problems. In particular, we describe a new coordination protocol that supports with application-level checkpointing. We have implemented this approach on a Windows 2000 cluster at the Cornell Theory Center. In Section 5, we discuss how we save and restore the state of the application and the MPI library. In Section 6, we measure the performance overheads of our approach by running a number of small benchmarks on this platform. The full paper will present more detailed measurements of these and larger benchmarks. We conclude in Section 7 with a discussion of future work.

2 Terminology

In this section, we introduce the terminology and notation used in the rest of the paper. Following usual practice, we assume that the system does not initiate the creation of a global checkpoint before all previous global checkpoints have been created and commited 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 3.

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.

[1] Note that MPI can bypass the protocol layer to read and write message buffers in the application space directly. Such manipulations, however, are not invisible to the protocol layer. MPI may not begin to access a message buffer until after it has been given specific permission to do so by the application (e.g. via a call to `MPI_Irecv`). Similarly, once the application has granted such permission to MPI, it should not access that buffer until MPI has informed it that doing so is safe (e.g. with the return of a call to `MPI_Wait`). The calls to, and returns from, those functions are intercepted by the protocol layer.

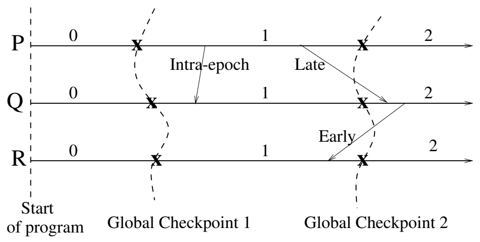

Figure 3: Epochs and message classification

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

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. In particular the name "*inconsistent* message" comes from the fact that a recovery line that is crossed by such a message represents a state where a message has been received but not yet sent, a situation that can never happen in a real execution. However, the fact that application-level checkpointing allows us to take checkpoints only at specific spots means that in this problem domain we cannot avoid recovery lines with *inconsistent* messages. Thus they are a completely natural feature of the problem and we renamed them to make this difference clear.

3 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.

3.1 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.

3.2 Handling late and early messages

We use Figure 3 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. 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.

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.

3.3 Non-FIFO message delivery at application level

Many system-level protocols assume that the communication between a pair of processes behaves in a FIFO manner. For example, in the Chandy-Lamport protocol, a process P that takes a checkpoint sends a marker token to other processes, informing them of what it has done. The protocol relies on the FIFO assumption to ensure that these other processes must receive this token before they can receive any message sent by P after it took its checkpoint.

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

3.4 Collective communication

The MPI standard includes collective communications functions such as `MPI_Bcast` and `MPI_Alltoall`, which involve the exchange of data among a number of processors. However, most checkpointing protocols in the literature, which were designed in the context of distributed computing, ignore the issue of collective communication.

The difficulty presented by such functions occurs when some processes make a collective communication call before taking their checkpoints, and others after. We need to ensure that on restart, the processes that reexecute the calls do not deadlock and receive correct information. Furthermore, `MPI_Barrier` guarantees specific synchronization semantics, which must be preserved on restart.

3.5 Problems Checkpointing MPI Library State

The key issue in performing application-level checkpointing of the state of the MPI library is that we do not assume to have access to its source code. While it would be possible for us to add application-level checkpointing methods to an existing MPI implementation, this would limit the portability of our checkpointer and would keep the programmer from using vendor-provided, platform-optimized implementations of MPI. Thus, our problem is to record and recover the state of the MPI library using only the MPI interface.

The library state can be broken up into three categories:

- **Library message buffers.** At the application-level, messages are invisible until they are received by the application. Therefore, at checkpoint time, the application cannot distinguish whether a given message is sitting in a network buffer on the sending processor, being transmitted, or sitting in a network buffer on the destination processor. All such messages are equivalently "in-flight" from the application's perspective. Therefore, we do not need to checkpoint the library's communication buffers.
- **MPI's *opaque* objects.** Such objects are internal to the MPI library but are visible to application may via handles. These objects include request objects (`MPI_Request`), communicators (`MPI_Comm`), groups (`MPI_Group`), data types (`MPI_Datatype`), error handlers (`MPI_Errhandler`), user defined operators (`MPI_Op`), and key-value pairs.
- **State internal to the MPI library.** There is certain state in the MPI library, such as message queues, timers and the network addresses of processors, that is completely hidden to the application. Since this state cannot be manipulated via MPI's interface, it is impossible for us to save or restore it. However, this is not required for correctness. All that is required is that the application's view of the library remains consistent before and after restart.

4 A Non-Blocking, Coordinated Protocol for Application-level Checkpointing

We now describe the coordination protocol for global checkpointing. The protocol is independent of the technique used by processes to take local checkpoints. To avoid complicating the presentation, we first describe the protocol for point-to-point communciation only. Then, we show that collective communication can be handled elegantly using the mechanism in place for point-to-point communication.

4.1 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 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.

4.2 Piggybacked information on messages

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

1. Is the message a late, intra-epoch, or early message?
2. Has the sending process stopped 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 can be piggybacked instead of the epoch number. With this optimization, the piggybacked information reduces to two booleans and an integer.

Further optimization is possible. If 32-bit integers are used, the two most significant bits of an integer can be used to represent the color of the epoch and the state of the *amLogging* flag of the sender, and remaining 30 bits can be used as the messageID. This solution should work fine because it is unlikely that a single process will send more than a billion messages between checkpoints! With this optimization, the protocol can be implemented by piggybacking a single integer on the application payload.

4.3 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 4, 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.

4.4 Putting it all together

Figure 4 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 ∞.

4.5 Collective Communication

We will use `MPI_Allreduce` to illustrate how collective communication is handled. In Figure 5, collective communication call A shows an `MPI_Allreduce` call in which processes P and Q execute the call after taking local checkpoints, and process R executes the call before taking the checkpoint. During recovery, processes P and Q will reexecute this collective communication call, but process R will not. Unless something is done, the program will not recover correctly.

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

A simple solution is to require a process to log the result of every collective communication call it makes during the

```
communicationEventHandler()
    Application message send to process d:
        Piggyback <epoch,amLogging,nextMessageID>
                on the message
        sendCount[d]++
        nextMessageID++
    Application message receive from process u:
        Remove <epoch_u,amLogging_u,messageID_u>
                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 ← true
    Control message: stopLogging
        finalizeLog()
    Control message: mySendCount(n) from process u
        totalSent[u] ← 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] ← ∞ for all senders u

finalizeLog()
    write log to stable storage
    amLogging ← false
    send StoppedLogging message to initiator

potentialCheckpoint()
    if (checkpointRequested = false) return
    save node state to stable storage (see Section 5)
    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] ← nil
    checkpointRequested ← false
    amLogging ← true
    nextMessageID ← 0
    receivedAll?()
```

Figure 4: Application-level Checkpointing Protocol

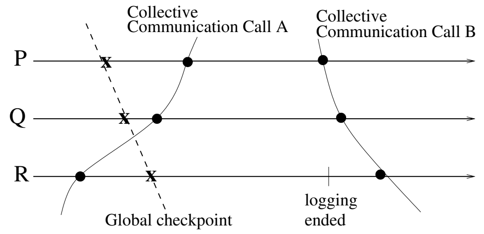

Figure 5: Collective Communication

time it is logging. Collective communication call B in Figure 5 illustrates a subtle problem with this solution - process R executes the `MPI_Allreduce` after it has stopped logging, so it would be incorrect for processes P and Q to log the results of their call. This problem is similar to the problem encountered in the point-to-point message case, and the solution is similar (and simpler). Each process piggybacks its *amLogging* bit on the application data, and the function invoked by `MPI_Allreduce` computes the conjunction of these bits. If any process involved in the collective communication call has stopped logging, all the other processes get to know about it, and do not log the result of the call; they also stop logging.

The elegance of this solution owes much to the decision to implement the protocol in a layer that sits between the application program and the MPI library. Each collective communication call is actually implemented by the MPI layer using many point-to-point messages. Had the layer been implemented between MPI and the operating system/hardware layer, the protocol would have had to deal with all these low-level point-to-point messages, which would be far more complex.

Most of the other collective communication calls can be handled in this way. Ironically, the only one that requires special treatment is `MPI_Barrier`. Suppose that the collective communication call A in Figure 5 is an `MPI_Barrier`. The solution described above will effectively convert the barrier to a no-op during recovery, which is incorrect since barriers are used to synchronize processes. The correct solution is to ensure that all processes involved in a barrier execute it in the same epoch. A simple implementation is the following. All processes involved in the barrier execute an all-to-all communication just before the barrier to determine if they are all in the same epoch. If not, processes that have not yet taken their local checkpoints do so, ensuring that the barrier is executed by all processes in the same epoch. This solution requires the precompiler to insert the all-to-all communication and the potential checkpointing calls before each barrier.

5 State Saving

5.1 Application state-saving

The state of the application running on each node consists of its position in the static text of the program, its position in the dynamic execution of the program, its local and global variables, and its heap-allocated structures. The precompiler modifies the application source so that this state is correctly saved, and can be restarted, at the `potentialCheckpoint` positions in the original code.

The approach that we describe does not currently save any less data than system-level checkpointing. However, it is a starting point for optimizing the amount of state that is saved at a checkpoint. In Section 7, we describe ongoing work towards this goal.

5.1.1 Checkpointing the application's position

Checkpointing a process' position is handled by inserting labels at the `potentialCheckpoint` and function call locations in the original source. We utilize a data structure, the *Position Stack* (*PS*) to record a trace of a program's execution by inserting code to manipulate the PS as labels are encountered. Figure 6 shows an example of the code inserted by the precompiler to manipulate the *PS*.

When a checkpoint is taken, the *PS* is saved as part of the checkpoint. If the application is restarted, the *PS* is restored, and each function jumps to the label that it stored on the *PS*. In such a manner, the activation stack is rebuilt and the program is prepared to resume immediately after the `potentialCheckpoint` location where the checkpoint was taken.

```
function1()
{
        if(restart)
                goto (PS.item(i++))
        //...
        PS.push(1);
label_1:
        function2();
        PS.pop();

        //...
        PS.push(2);
        potentialCheckpoint();
label_2:
        PS.pop();
        //...
}
```

Figure 6: *Position Stack* manipulation

The precompiler only needs to insert labels at function calls that can eventually lead to a `potentialCheckpoint` location. In order to insure that the *PS* correctly reflects which function call is currently active, the precompiler needs to decompose certain complex statements, such as a statement containing two calls to checkpointable functions, or a return statement that makes a call to one.

5.1.2 Checkpointing the application's data

If we ensure that the processes' original and recovered stack always begins at the save virtual address, using the techniques described above will ensure that, after restart, the activation stack frames will have same positioning as during the original run. Therefore, a stack variable will have the same virtual address both before and after restart.

We utilize another data structure, the *Variable Descriptor Stack VDS* to save and restore the stack variables' values. The *VDS* stores the address and size of each stack variable. The precompiler inserts code that manipulates this structure as variables enter and leave scope. Figure 7 shows such manipulations.

```
function(int a)
{
        VDS.push(&a, sizeof(a));
        int b[10];
        VDS.push(&b, sizeof(b));
        {
                int c;
                VDS.push(&c, sizeof(c));
                //...
                VDS.pop;
        }
        VDS.pop;
        VDS.pop;
}
```

Figure 7: Manipulating the *VDS*

The application uses the *VDS* to save and restore the stack variables' values. When a checkpoint is taken, for every record in the *VDS*, it copies the specified number of bytes, from the specified address, into the checkpoint file. On restart, we first restore the stack using the *PS*, and then use the *VDS* to restore stack variables by copying their value from the checkpoint to their locations on the stack. The *VDS* must be saved and restored as part of the local checkpoint.

A similar mechanism can be used to handle global variables. In order to discover all of a program's global variable, either the precompiler must have access to all source files of the program at once, or this discovery must be done during linking. We are currently using the former approach.

5.1.3 Checkpointing the application's heap

Similar to the stack variables, a heap allocated object, upon restart, needs to be restored to the same virtual address that it had in the original process. Additionally, we would also need to ensure that the heap management structures (ie. the free list) are restored correctly. Therefore, our precompiler provides its own heap management system.

This heap management system maintains a *Heap Object Structure*, *HOS*, which is similar to the *VDS* and contains the starting address and length of each "live" heap object. When checkpointing, we use the *HOS* to copy the heap objects to the checkpoint file. The *HOS*, along with some other heap management structures, is saved with the checkpoint. On restart, we request the same chunk of virtual address space, restore the *HOS*, and use it to copy the objects from the checkpoint file back onto the heap.

5.1.4 A note on pointers

Because stack variables and heap objects are restored to their original virtual addresses, we need to make no special consideration regarding data pointers: they are saved as ordinary data. A valid data pointer in the original process will point to the same object in the recovered one.

This strategy differs significantly from the one used in the PORCH ([15]). Because their goal was to create a checkpoint file that could be used within a heterogeneous environment, they could make no assumptions regarding the address or length of a program's variables. Instead they were forced to employ "re-locatable" pointers and to convert values to an architecture neutral representation when checkpointing.

The disadvantages to such techniques are that a programmer is required to work with a subset of the C language that disallows arbitrary casting, and that there is a performance cost to be paid when converting values from one representation to another. Since portability is not one of our goals, and because we feel that the limitations on programming style and the added overhead of doing pointer conversion are too burdensome for our applications, we have chosen not to follow the PORCH approach.

5.2 MPI Library State-Saving

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

The MPI opaque objects whose handles are stored in the pseudo-handles can be divided into two types: *transient* and *persistent*. Transient objects come into existence often and tend to have short lifetimes while persistent objects come into existence rarely and tend to have long lifetimes. We use a separate mechanism for reinitializing the pseudo-handles of each type of MPI opaque object.

The only MPI objects that we consider as transient are `MPI_Request` objects. These objects are created by non-blocking communication functions, such as `MPI_Isend` or `MPI_Irecv`, and are destroyed by functions such as `MPI_Wait`. When a `MPI_Isend` or `MPI_Irecv` that creates a `MPI_Request` object occurs before a checkpoint and the the call to `MPI_Wait` that destroys the object occurs after the checkpoint, then on recovery, the pseudo-handle for that `MPI_Request` object must be correctly reinitialized. This does not necessarily mean that the `MPI_Request` object must be recreated; it means that calling `MPI_Wait` with the pseudo-handle must have the same effect that it did during the original execution.

The pseudo-handle for an `MPI_Request` object created by `MPI_Isend` must be reinitialized so that the call to `MPI_Wait` will return immediately, which means that the send buffer may be reused by the application. This is because the call to `MPI_Isend` that created the request objected occurred before the checkpoint. Either the message was received before the receiving processor tooks its checkpoint, in which case the data is part of the checkpoint, or after, in which case the message is stored in the receiver's logs. In either case, it is safe for the application to reuse the buffer.

The pseudo-handle for an `MPI_Request` object created by `MPI_Irecv` must be reinitialized in one of two ways. If the receive matches a late message in the receiver's log, this message may be copied to the receiver buffer and `MPI_Wait` may return immediately. If the receive does not match any late message, then it must match a send that is issued after checkpointing. In this case, on recovery, `MPI_Irecv` must be called again with exactly the same arguments and its handle stored in the pseudo-handle.

All objects besides `MPI_Request`'s are classified as persistent opaque objects and are handled as follows. Each processor records all the function names and arguments of every call that creates or manipulates these persistent objects. This record is saved to stable storage as part of the local checkpoint. On restart, each processor will replay these calls in order to recreate effectively the same persistent objects that existed at the time of the checkpoint. The pseudo-handles are reinitialized with the handles to these new objects.

6 Performance

6.1 Experimental setup

We performed our experimental evaluation on the CMI cluster at the Cornell Velocity supercomputer. This cluster is composed of 64 2-way PentiumIII 1Ghz nodes, featuring

2GB of RAM and connected by a Giganet switch. The nodes have 40MB/sec bandwidth to local disk. *Due to hardware problems, we used only 16 of those processors for our tests; in the final paper, we will present results for the full machine.* The operating system on the machines was Windows 2000 and we used MPI/Pro 1.6.4 as our MPI implementation. The applications were compiled using the Microsoft C/C++ Optimizing Compiler version 12, using the "Optimized for Speed" optimization setting. We evaluated the performance of our checkpointer on three codes:

- A dense Conjugate Gradient code from Yingfeng Su of the University of San Francisco. This code implements a parallel conjugate gradient algorithm with block row distribution. The main loop performs a parallel matrix vector multiply and a parallel dot product, with communication coming from an allReduce and an allGather, which are implemented in terms of point-to-point messages along a butterfly tree. We ran the dense CG code for 500 iterations.
- A Laplace Solver, by Raghu Reddy from the Pittsburgh Supercomputing Center. This program uses a $n \times n$ grid of numbers that is distributed by block rows. During each iteration every grid cell is updated to be the average of the numbers contained by the neighboring cells (up, down, left, right) in the previous iteration. The communication comes from each processor exchanging border rows with the processor "above" it and the processor "below" it. We ran the Laplace code for 40000 iterations.
- Neurosys, a neuron simulator by Peter Pacheco of the University of San Francisco (available publically at http://nexus.cs.usfca.edu/neurosys/), uses a graph of neurons which excite and inhibit each other via their connections. The current state of each neuron is computed by solving a function of the states of the neurons that are connected to it. The evolution of the neuron network through time is computed via the Runge-Kutta method for differential equations. The program is parallelized by assigning each processor a block of neurons to work with. Communication consists of 5 `MPI_Allgather`'s and 1 `MPI_Gather` in each loop iteration. We ran Neurosys for 3000 iterations.

All the checkpoints in our experiments are written to the local disk, with a checkpoint interval of 30 seconds.

6.2 Performance

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

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

Experimental results are shown in Figure 8.

- In dense CG, the total overhead for taking full checkpoints every 30 seconds is 14% for a 4096x4096 or 8192x8192 matrix. This increases dramatically to 43% when we move up to 16384x16384. However, since the overhead is only 4.5% when we do everything but record the application state, it is clear that the reason for the increased overhead is that size of application state.
- The addition of checkpointing to the Laplace Solver adds only 2.1% overhead in the worst case tested. This can be explained by the fact that even biggest data set we tested had only 2.1MB of application state, which is much less than the amount where the dense conjugate gradient code began slowing down. Furthermore, the amount of data the Laplace Solver sends per message is much more than the data that we attach to each message, so our piggybacked information adds little overhead.
- Neurosys does a lot of computation and communication on a relatively small data set. Its small application state, which varies from 18KB to 1.24MB, is too small to cause much overhead from recording the application state. However, we see another interesting overhead in the difference between the runtimes of the unmodified version and the version that uses the protocol layer but takes no checkpoints. The primary difference between the two is that the latter piggybacks data on messages. Neurosys uses 5 `MPI_Allgather`'s in every iteration and in our implementation, each such data `MPI_Allgather` is preceeded by a command `MPI_Allgather` which sends around the relevant control information. This accounts for the jump in runtime which is as high as 160% for 16x16. However, as the input sizes increases, the message sizes and computation time also increase but the number of messages does not. Thus, the additional work masks the overhead associated with passing around control data, leading this overhead to drop to 85% of the total runtime for 32x32, 34% for 64x64 and just 2.7% for 128x128.

7 Conclusions and Future Work

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

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

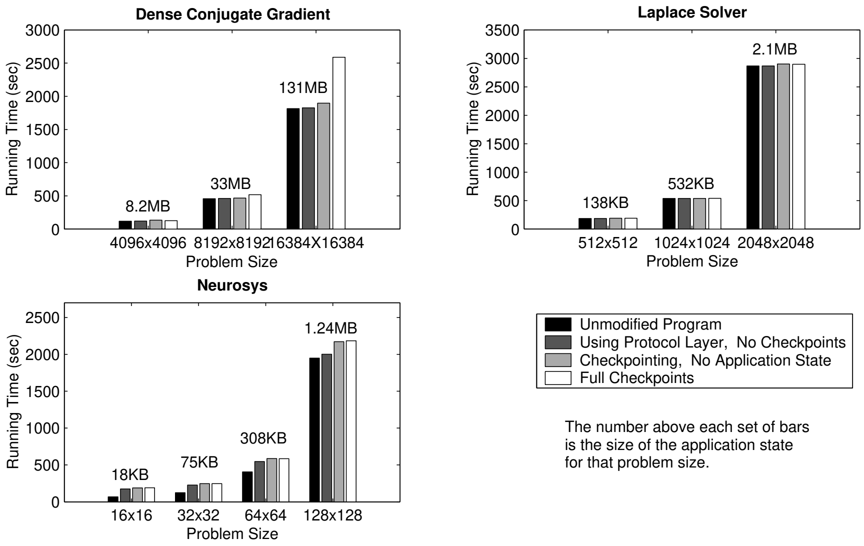

Figure 8: Performance Charts

reconstructed by the implementation of our protocol.

The goal of our project is to provide a highly efficient checkpointing mechanism for MPI applications. One way to minimize checkpoint overhead is to reduce the amount of data that must be saved when taking a checkpoint. We are continuing the development of our precompiler so that it may utilize analysis techniques to determine areas of memory that can be safely excluded from a checkpoint.

Others have worked on using compiler technologies to avoid checkpointing dead and read-only variables [2]. Their work focussed on statically allocated data structures in FORTRAN programs. We would like to extend such work to handle the dynamically created in C/MPI applications.

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

Both these techniques are actually specializations of a more general technique that we term *recomputation checkpointing*. For some data structures, a compiler might be able to determine how to recompute their values. If the description of this recomputation requires less space than storing their data, we should store the description, rather than the data, in the checkpoint.

We would also like to extend this work to provide fault-tolerance for other types of high performance computing systems, such as shared memory machines, and the MPI-2 message passing standard.

References

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

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

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

[4] 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. 1992.

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

[7] R. Graham, S.-E. Choi, D. Daniel, N. Desai, R. Minnich, C. Rasmussen, D. Risinger, and M. Sukalski. A network-failure-tolerant message-passing system for tera-scale clusters. In *Proceedings of the International Conference on Supercomputing 2002*, 2002.

[8] I. Gupta, T. Chandra, and G. Goldszmidt. On scalable and efficient distributed failure detectors, 2001.

[9] IBM Research. Blue gene project overview. Online at http://www.research.ibm.com/bluegene/, 2002.

[10] D. B. Johnson and W. Zwaenepoel. Transparent optimistic rollback recovery. *Operating Systems Review*, 25(2):99–102, 1991.

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

[12] 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.

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

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

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

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

[17] T. Tabe and Q. F. Stout. The use of the MPI communication library in the NAS parallel benchmarks. Technical Report CSE-TR-386-99, 17, 1999.

[18] The BlueGene/L Team. An overview of the bluegene/l supercomputer. In *SC 2000 High Performance Networking and Computing*, 2002.