The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing

Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine
Open Systems Laboratory, Indiana University
{ssankara,jsquyres,brbarret,lums}@lam-mpi.org

Jason Duell, Paul Hargrove, Eric Roman
Lawrence Berkeley National Laboratory
{jcduell,phhargrove,eroman}@lbl.gov

Abstract

As high-performance clusters continue to grow in size and popularity, issues of fault tolerance and reliability are becoming limiting factors on application scalability. To address these issues, we present the design and implementation of a system for providing coordinated checkpointing and rollback recovery for MPI-based parallel applications. Our approach integrates the Berkeley Lab BLCR kernel-level process checkpoint system with the LAM implementation of MPI through a defined checkpoint/restart interface. Checkpointing is transparent to the application, allowing the system to be used for cluster maintenance and scheduling reasons as well as for fault tolerance. Experimental results show negligible communication performance impact due to the incorporation of the checkpoint support capabilities into LAM/MPI.

1 Introduction

In recent years, the supercomputing community has seen a significant increase in the CPU count of large-scale computational resources. Seven of the top ten machines in the November 2002 Top500 [1] list utilize at least 2000 processors. With machines such as ASCI White, Q, and Red Storm, the processor count for the largest systems is now on the order of 10,000 processors—and this increasing trend will only continue. While the growth in CPU count has provided great increases in computing power, it also presents significant reliability challenges to applications. In particular, since the individual nodes of these large-scale systems are comprised of commodity hardware, the reliability of the individual nodes is targeted for the commodity market. As the node count increases, the reliability of the parallel system decreases (roughly proportional to the node count). Indeed, anecdotal evidence suggests that failures in the computing environment are making it more difficult to complete long-running jobs and that reliability is becoming a limiting factor on scalability.

The Message Passing Interface (MPI) is a *de facto* standard for message passing parallel programming for large-scale distributed systems [12, 14, 16, 17, 24, 30]. Implementations of MPI comprise the middleware layer for many large-scale high-performance applications [3, 15, 18, 37]. However, the MPI standard itself does not specify any particular kind of fault tolerant behavior. In addition, the most widely used MPI implementations have not been designed to be fault-tolerant.

To address these issues, we present the design and implementation of a system for providing coordinated checkpointing and rollback recovery for MPI-based parallel applications. Several factors were considered for our design.

Generality. Our design is an extension of the component framework comprising the most recent version of LAM/MPI [32, 33]. In general, the framework itself can be used to support a wide variety of fault tolerance mechanisms; we report on one such mechanism here. In particular, our approach integrates the Berkeley Lab BLCR kernel-level process checkpoint system with the LAM implementation of MPI through a defined checkpoint/restart interface.

Transparency. The particular implementation of coordinated checkpointing and rollback recovery that we report here was designed with transparency in mind. That is, our system can be used to checkpoint parallel MPI applications without making any changes to the application code. Involuntary checkpointing is consequently supported.

Performance. As shown by our experimental results, the addition of checkpointing support capabilities to LAM/MPI has insignificant impact on its message passing performance. And, since checkpoint support is run-time selectable, it can be bypassed altogether for applications that do not wish to use it.

Portability. Our implementation has been incorporated into the most recent release of LAM/MPI, a widely used and industrial strength open-source implementation of MPI. Although the BLCR checkpointer is currently available for Linux, LAM/MPI will operate on almost all POSIX systems. The general approach taken in this work will allow it to be easily extended to other single process checkpoint systems and to other operating systems.

The remainder of the paper is organized as follows. Section 2 discusses background information and related work. The design of our system is given in Section 3 and details of its implementation in Section 4. Performance results are provided in Section 5. Future work and our conclusions are given in Sections 6 and 7.

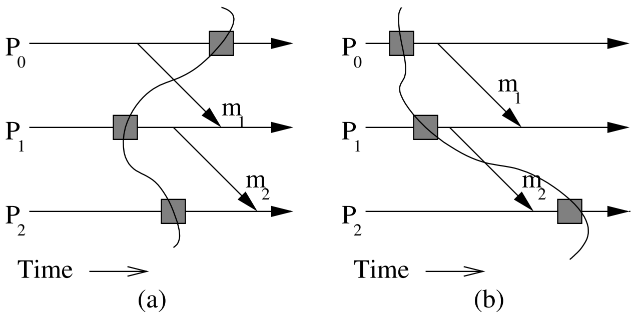

Figure 1: A message-passing system consisting of 3 processes. (a) shows an example of a consistent global state where message m_1 is recorded as having been sent by process P_0 but not yet received by process P_1, and (b) shows an example of an inconsistent global state in which message m_2 is recorded as having been received by P_2 but not yet sent by P_1.

2 Background

2.1 Checkpoint-Based Rollback Recovery

In the context of message-passing parallel applications, a *global state* is a collection of the individual states of all participating processes and of the states of the communication channels. A *consistent global state* is one that may occur during a failure-free, correct execution of a distributed computation. Within a consistent global state, if a given process has a local state that indicates a particular message has been received, then the state of the corresponding sender must indicate that the message has been sent [4]. Figure 1 shows two examples of global states, one of which is consistent, and the other of which is inconsistent. A *consistent global checkpoint* is a set of local checkpoints, one for each process, forming a consistent global state. Any consistent global checkpoint can be used to restart process execution upon failure.

Checkpoint/restart techniques for parallel jobs can be broadly classified into three categories: uncoordinated, coordinated, and communication-induced. (These approaches are analyzed in detail in [10].)

2.1.1 Uncoordinated Checkpointing

In the uncoordinated approach, the processes determine their local checkpoints independently. During restart, these processes search the set of saved checkpoints for a consistent state from which execution can resume. The main advantage of this autonomy is that each process can take a checkpoint when it is most convenient. For efficiency, a process may take checkpoints when the amount of state information to be saved is small [39]. However, this approach has several disadvantages. First, there is the possibility of the *domino effect* [26] which causes the system to rollback to the beginning of computation, resulting in the loss of a large amount of useful work. Second, a process may take checkpoints that will never be part of a global consistent state. Third, uncoordinated checkpointing forces each process to maintain multiple checkpoints, thereby incurring a large storage overhead.

2.1.2 Coordinated Checkpointing

With the coordinated approach, the determination of local checkpoints by individual processes is orchestrated in such a way that the resulting global checkpoint is guaranteed to be consistent [4, 9, 19, 35, 38]. Coordinated checkpointing simplifies recovery from failure and is not susceptible to the domino effect, since every process always restarts from its most recent checkpoint. Also, coordinated checkpointing minimizes storage overhead since only one permanent checkpoint needs to be maintained on stable storage. The main disadvantage of coordinated checkpointing, however, is the large latency involved in saving the checkpoints, since a global checkpoint needs to be determined before the checkpoints can be written to stable storage.

2.1.3 Communication-Induced Checkpointing

The communication-induced checkpointing approach forces each process to take checkpoints based on protocol-related information piggybacked on the application messages it receives from other processes [27]. Checkpoints are taken such that system-wide consistent state always exists on stable storage, thereby avoiding the domino effect [2]. Processes are allowed to take some of their checkpoints independently. However, in order to determine a consistent global state, processes may be forced to take additional checkpoints. The checkpoints that a process takes independently are called local checkpoints, while those that a process is forced to take are called forced checkpoints. The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint. The forced checkpoint must be taken before the application may process the contents of the message, possibly incurring high latency and overhead. In contrast with coordinated checkpointing, no special coordination messages are exchanged in this approach.

2.2 Other Uses of Checkpoint/Restart

The ability to checkpoint and restore applications has a number of uses in a parallel environment besides fault tolerance.

Gang scheduling—checkpointing and restarting all the processes that are part of a single parallel application—allows for more flexible scheduling. For example, jobs with large resource requirements can be intermittently scheduled at off-peak times using the checkpoint/restart capability. Without intermittent scheduling such large jobs may use all available resources for long periods—locking out other jobs during that time. Hence, the ability to stop and resume large jobs allows scheduling of other available jobs in such a way that the overall system throughput is maximized.

Process migration is another feature that is made possible by the ability to save a process image. If a process needs to be moved from one node to another (because imminent failure of a node is predicted or for scheduling reasons) it is possible to transfer the state of the processes running on that node to another node by writing the process image directly to a remote node. The process can then resume execution on this new node, without having to kill the entire application and start it all over again. Process migration has also proved extremely valuable for systems whose network topology constrains the placement of processes in order to achieve optimal performance. The Cray T3E's interconnect, for instance, uses a three-dimensional torus that requires processes that are part of the same parallel application to be placed in contiguous locations on the torus. This results in fragmentation as jobs of different sizes enter and exit the system. With process migration, jobs can be packed together to eliminate fragmentation, resulting in significantly higher utilization [40]. Networks with such constraining topologies have become less common recently, however IBM's Blue Gene/L project plans to constrain communication among processors [36], and more cluster projects may use them in the future.

2.3 Related Work

Checkpoint/restart for sequential programs has been somewhat well studied. Libckpt [25] is an open source library for transparent checkpointing of Unix processes. It contains support for incremental checkpoints, in which only pages that have been modified since the last checkpoint are saved. Condor [22, 23] is another system that provides checkpointing services for single process jobs on a number of Unix platforms. The CRAK (Checkpoint/Restart As a Kernel module) project [41] provides a kernel implementation of checkpoint/restart for Linux. CRAK also supports migration of networked processes by adopting a novel approach to socket migration. BLCR (Berkeley Lab's Checkpoint/Restart) [8] is a kernel implementation of checkpoint/restart for multi-threaded applications on Linux. Libtckpt [7] is a user-level checkpoint/restart library that can also checkpoint POSIX threads applications.

In the context of parallel programs, there are vendor implementations of checkpoint/restart for MPI applications running on some commercial parallel computers [6]. Some implementations are also available for checkpointing MPI applications running on commodity hardware. CoCheck [34] is one such tool for PVM and MPI applications. It is built into a native MPI library called tuMPI and layered on top of a portable single-process checkpointing mechanism [13, 21]. CoCheck uses a special process to coordinate checkpoints, that sends a checkpoint request notification to all the processes belonging to the MPI job. On receiving this trigger, each process sends a "ready message" (RM) to all other processes, and stores all incoming messages from each process until all the RMs have been received, in specially reserved buffers. The underlying checkpointer then saves the execution context of each process to stable storage. At restart, a receive operation first checks the buffers for a matching message. If there is such a message, it is retrieved from the buffer. Otherwise, a real receive operation fetches the next matching message from the network. One drawback to CoCheck is that a checkpoint request cannot be processed when a send operation is in progress. Consequently, if a matching receive has not been posted by the peer, there is no finite bound on the time taken for the checkpoint request to complete. Also, checkpointing could change the semantics of MPI's synchronous sends in CoCheck: an anticipated receive could cause the return of the send instead of the actual receive by the application.

A checkpoint/restart implementation for MPI at NCCU Taiwan uses a combination of coordinated and uncoordinated strategies for checkpointing MPI applications [20]. It is built on top of the NCCU MPI implementation [5], and uses Libckpt as the back-end checkpointer. Checkpointing of processes running on the same node is coordinated by a local daemon process, while processes on different nodes are checkpointed in an uncoordinated manner using message logging.

A limitation of the existing systems for checkpointing MPI applications on commodity clusters is that they are implemented using MPI libraries that primarily serve as research platforms and are not widely used. Another drawback of some of these checkpoint/restart systems is that they are tightly coupled to a specific single-process checkpointer. Since single-process checkpointers usually support a limited number of platforms, this limits the range of systems on which MPI applications can be checkpointed to those that are supported by the underlying checkpointer.

3 Design

This section presents an overview of the design of the checkpoint/restart system in LAM/MPI. This implementation does not alter the semantics of any of the MPI functions, and fully supports all of MPI-1. The checkpoint/-restart system has been designed in such a way that there is a clear separation between the checkpoint/restart functionality and MPI-specific functionality in LAM. Also, the checkpoint/restart system can "plug-in" multiple back-end checkpointers with minimal changes to the main LAM/MPI code base, as a result of which there is a wide range of platforms that can potentially be supported by our system. The current implementation in LAM/MPI uses the BLCR [8] checkpointer that is available for Linux.

3.1 Checkpointing Approach in LAM/MPI

A checkpoint of an MPI job is initiated by a user or a batch scheduler by delivering a checkpoint request to `mpirun`. The precise mechanism for delivering this request is implementation-dependent. On receiving this request, `mpirun` propagates this request to all the processes in the MPI job.

LAM/MPI uses a coordinated approach to checkpointing MPI jobs. The current implementation in LAM supports a TCP-based communication sub-system (see Sections 3.2.1 and 4). Upon receiving the checkpoint request from `mpirun`, all the MPI processes interact with each other to guarantee that their local checkpoints will result in a consistent global checkpoint. In [4], a consistent global state is described as the set of process states and the states of their communication channels. The approach adopted in LAM ensures that all the MPI communication channels between the processes are empty when a checkpoint is taken. During restart, all the processes resume execution from their saved states, with the communication channels restored to to their known (empty) states.

```
void bookmark_exchange() {
  int i;
  struct bookmark *bookmarks_arr;

  for (i = (num_procs − myidx − 1), j = 0; j < num_procs;
       i = (i + 1) % num_procs, ++j) {
    if (myidx > i) {
      /* send our bookmark status, then receive into
         appropriate location in bookmarks array */
      send_bookmarks(i);
      recv_bookmarks(bookmarks_arr);
    } else if (myidx < i) {
      /* receive remote bookmark status into appropriate
         location in bookmarks array, then send */
      recv_bookmarks(bookmarks_arr);
      send_bookmarks(i);
    }
  }
}
```

Figure 2: Staggered all-to-all algorithm used for communicating network status.

The interaction between the processes to clear the data in the MPI communication channels uses a "staggered all-to-all" algorithm over out-of-band communication channels that are available in LAM, as shown in Figure 2. This algorithm starts with each process choosing a unique peer to exchange information about how much data it has sent to and received from that peer. This exchange then continues with other peers in increasing order of ranks in a circular fashion until each process has exchanged this information with its immediate lower-ranked peer. Then, based on this information, each process receives the remaining data from the MPI communication channels and all the in-flight data are drained.

The LAM checkpoint algorithm is summarized below. `mpirun` acts as a coordination point between all processes of an MPI application, and is the process signaled by the run-time system or user when a checkpoint is to be initiated.

1. **mpirun:** receives a checkpoint request from a user or batch scheduler.
2. **mpirun:** propagates the checkpoint request to each MPI process.
3. **mpirun:** indicates that it is ready to be checkpointed.
4. **each MPI process:** coordinates with the others to reach a consistent global state in which the MPI job can be checkpointed. For example, processes using

TCP for MPI message passing drain in-flight messages from the network to achieve a consistent global state.

5. **each MPI process:** indicates that it is ready to be individually checkpointed.
6. **underlying checkpointer:** saves the execution context of each process to stable storage.
7. **each MPI process:** continues execution after the checkpoint is taken.

The following sequence of events occurs at restart:

1. **mpirun:** restarts all the processes from the saved process images.
2. **each MPI process** sends its new process information to `mpirun`
3. **mpirun:** updates the global list containing information about each process in the MPI job and broadcasts it to all processes.
4. **each MPI process:** receives information about all the other processes from `mpirun`.
5. **each MPI process:** re-builds its communication channels with the other processes.
6. **each MPI process:** resumes execution from the saved state.

This algorithm has been successfully implemented using the BLCR [8] checkpointer. The details of the implementation are given in Section 4.

3.2 LAM/MPI Architecture

LAM/MPI is designed with two major layers: the LAM layer and the MPI layer, as shown in Figure 3. The LAM layer provides a framework and run-time environment upon which the MPI layer executes. The LAM layer provides services such as message passing, process control, remote file access, and I/O forwarding. The MPI layer provides the MPI interface and an infrastructure for direct, process-to-process communication over high-speed networks.

LAM provides a daemon-based run-time environment (RTE). A user-level daemon (the `lamd`) is used to provide many of the services needed for the MPI RTE. The `lamboot` command is used to start a `lamd` on every node at the beginning of an execution. At the end of an execution session, these `lamd`s are killed using the `lamhalt` command.

The `lamd`s provide process control for all MPI jobs executed under LAM/MPI. `mpirun` launches an MPI application by sending a request to the appropriate daemons, which in turn `fork()`/`exec()` the application. When an application terminates, the daemons are notified through the standard Unix `SIGCHLD` mechanisms, and they relay this information back to `mpirun`. The LAM daemons also provide message-passing services over UDP channels.

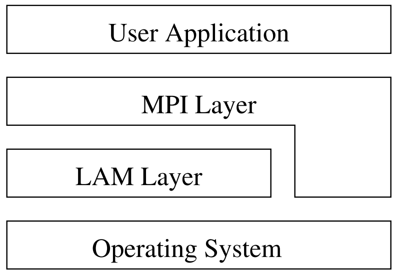

Figure 3: The layered design of LAM/MPI.

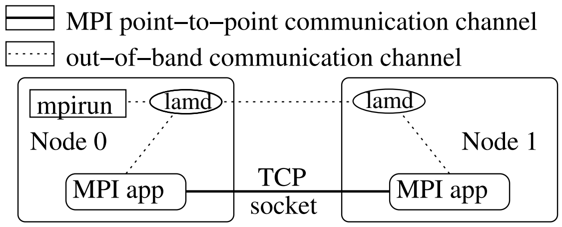

Figure 4: A two-way MPI job on two nodes.

The MPI library consists of two layers. The upper layer is portable and independent of the communication subsystem (i.e., MPI function calls and accounting utility functions). The lower layer consists of a modular framework for components called SSI (see Section 3.2.1). One such component type is the MPI Request Progression Interface (RPI), which provides device-dependent point-to-point message-passing between the MPI peer processes. LAM/MPI includes RPIs that implement message-passing using TCP, shared memory, gm (the low-level message-passing system for Myrinet networks), and the message-passing service provided by the `lamd`s. Figure 4 shows the LAM/MPI RTE for a two-way MPI job running on two nodes and using the TCP RPI.

3.2.1 System Services Interface

LAM/MPI has recently been redesigned to provide a component framework for various services provided by the LAM infrastructure. This framework — the System Services Interface (SSI) — is composed of a number of component types, each of which provides a single service to the LAM RTE or MPI implementation [32]. Each SSI type can have one or more run-time selectable instances available. Component instances are implemented as plug-in modules, and are chosen at run-time, either automatically by the SSI infrastructure or manually by the user, allowing a particular version of LAM/MPI to support multiple underlying in-

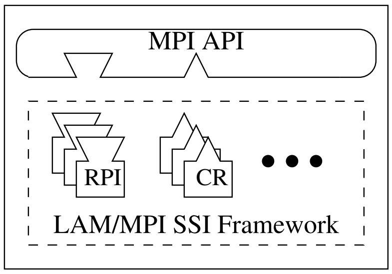

Figure 5: The LAM SSI component architecture has multiple different component types. At run-time, module instances will be chosen from each component type.

frastructures. Currently, there are SSI interfaces for launching the LAM RTE, MPI device-dependent point-to-point communication layer, MPI collective communication algorithms, and checkpoint/restart of MPI applications. Figure 5 shows the SSI framework, and how an MPI application can choose between modules of each component type at run-time.

The two component types shown in Figure 5 are the Request Progression Interface (RPI), and checkpoint/restart (CR). The RPI component type is responsible for all MPI point-to-point communications. The CR component type is the sole interface to the back-end checkpointing system to actually perform checkpoint and restart functionality.

Although LAM has multiple RPI modules available for selection at run-time, there is currently only one CR module available: `blcr`, which utilizes the BLCR single-node checkpointer (see Section 3.3). The design and implementation of the CR SSI and the `blcr` module were the main focuses of this work.

For an MPI job to be checkpointable, it must have a valid CR module and each of the other SSI modules that it has chosen at run-time must support some abstract checkpoint/-restart functionality. The internal SSI checkpoint/restart interfaces were carefully designed to preserve strict abstraction barriers between the CR SSI and the other SSI modules. Hence, the strict separation of back-end checkpointing services and communication allows new back-end checkpointing systems to be "plugged-in" simply by providing a new CR SSI module; the existing RPI modules (and other SSI component types) will be able to utilize its services with no modifications.

3.2.2 The CR SSI

At the start of execution of an MPI job, the SSI framework chooses the set of modules from each SSI component type that will be used. In the case of the CR SSI, it determines whether checkpoint/restart support was requested, and if so, a CR module is selected to run (in this case, it is `blcr` since it is the only module available).

All modules in the CR SSI provide a common set of APIs to be used by the MPI layer and another set of APIs that can be used by `mpirun`. The detailed design of the CR SSI component type is described in [28]. Broadly, these APIs provide the following functionality:

- **initialize:** used by the MPI layer to attach to the underlying checkpointer, and register callback(s) that will be invoked at checkpoint.
- **suspend:** used by the MPI application thread to suspend execution when it is interrupted by the callback thread (see Section 4.1).
- **disable checkpoint:** used by `mpirun` to enter a critical section during which it cannot be interrupted by a checkpoint request.
- **enable checkpoint:** used by `mpirun` to exit a critical section and allow incoming checkpoint requests.
- **finalize:** used by the MPI layer to perform cleanup actions and detach from the underlying checkpointer.

Most of the work in the CR SSI is done in a separate thread to allow preparation for checkpoint to happen asynchronously without blocking the execution of the main application thread. In the `blcr` module, this thread is created by the BLCR checkpointer when a callback is registered during the module's **initialize** action. However, the design of the CR SSI type does not require the underlying checkpointer to provide this thread. If a checkpointer does not implicitly provide a separate thread for callbacks, the module itself can create this extra thread during **initialize** and block its execution until a checkpoint request arrives. This design strategy serves to reduce the requirements imposed on the underlying checkpointing systems, thereby potentially increasing the range of checkpointers that can be supported.

3.2.3 The RPI SSI

To support checkpointing, an RPI module must have the ability to generically "prepare for checkpoint," "continue after checkpoint," and "restore from checkpoint". A checkpointable RPI module must therefore provide API functions to perform this functionality. The following functions will be invoked from the thread-based callback in the CR SSI:

- **checkpoint:** invoked when a checkpoint request comes in, usually to consume any in-flight messages.
- **continue:** invoked to perform any operations that might be required when a process continues execution after a checkpoint is taken.

- **restart:** invoked to re-establish connections and any other operations that might be required when a process restarts execution from a saved state.

Note that these functions are independent of which back-end checkpointing system is used; for example, the actions required for the TCP RPI to checkpoint, continue and restart are the same regardless of which CR SSI module is selected. The detailed design of the RPI SSI is described in [31].

3.3 The BLCR Checkpointer

The Berkeley Lab's Linux Checkpoint/Restart project (BLCR) [8] is a robust, kernel-level checkpoint/restart implementation. It can be used either as a stand-alone system for checkpointing applications on a single node, or by a scheduling system or parallel communication library for checkpointing and restarting parallel jobs running on multiple nodes. BLCR is implemented as a Linux kernel module (for recent 2.4 versions of the kernel, such as 2.4.18) and a user-level library. A kernel module implementation has the benefit that it allows BLCR to be easily deployed by new users without requiring them to patch, recompile, and reboot their kernel. While the current implementation of BLCR only supports checkpointing of single processes (including multi-threaded processes), checkpointing of process groups, sessions, and a full range of Unix tools will be supported in the future.

BLCR provides a simple user-level interface to libraries/applications that need to interact with checkpoint/-restart. It provides a mechanism to register user-level callback functions that are triggered whenever a checkpoint occurs, and that continue when the process restarts (or a periodic checkpoint for backup purposes completes). Two kinds of callbacks can be registered: signal-based callbacks that execute in signal-handler context, and thread-based callbacks that execute in a separate thread. These callbacks allow the application to shutdown its network activity (and perform analogous actions on some other uncheckpointable resource) before a checkpoint is taken, and restore them later. Callbacks are designed to be written as shown in Figure 6.

BLCR also provides user-level code with "critical sections" to allow groups of instructions to be performed atomically with respect to checkpoints. This allows the applications to ensure that special cases such as network initialization are not interrupted by a checkpoint. In some cases, such atomicity is not merely a matter of convenience but is vital for correct program operation.

```
void callback(void *data_ptr) {
  struct my_data *pdata = (struct my_data *) data_ptr;
  int did_restart;

  /* Do checkpoint-time shutdown logic */

  /* Tell system to take the checkpoint */
  did_restart = cr_checkpoint();

  if (did_restart) {
    /* Actions to restart from a checkpoint */
  } else {
    /* Actions to continue after a checkpoint */
  }
}
```

Figure 6: Template for signal-based and thread-based callback functions. The state of the entire process (including the callback's execution) is saved in the `cr_checkpoint` call, and restored at restart or after checkpoint is complete.

4 Implementation Details

The checkpoint/restart implementation in LAM/MPI relies on the availability of a message-passing service provided by the LAM layer. This service is used for out-of-band signaling and communication between the processes during checkpoint and restart. Although they play an important role during a checkpoint, the `lamd`s are not a logical part of an MPI application, and are themselves not checkpointed. The design of this system also presupposes the availability of a threads package on the target platform. Currently, support for checkpoint/restart has been implemented only for a modified version of the TCP RPI. However, this functionality will soon be extended to include all the RPIs. This section describes the details of the implementation in the context of the sequence of steps that occur in the system during checkpoint, upon continuing from a checkpoint, and when restarting from saved context.

4.1 Checkpoint

Since `mpirun` is the startup coordination point for MPI processes, it was the natural choice to serve as the entry point for a checkpoint request to be sent to a LAM/MPI job. At the start of execution, `mpirun` invokes the initialization function of the `blcr` checkpoint/restart SSI module to register both thread-based and signal-based callback functions with BLCR. The thread-based callback is required to propagate the checkpoint requests to the MPI processes. This cannot be done in signal context because the propagation of the checkpoint request uses some non-reentrant C library calls, and the use of non-reentrant functions from signal context can cause deadlocks.

When a checkpoint request is sent by a user or batch scheduler (by invoking the BLCR utility `cr_checkpoint` with the process ID of `mpirun`), it triggers the callbacks to start executing. The thread-based callback computes the names under which the images of each MPI process will be stored to disk and saves the process topology of the MPI job (called "application schema" in LAM) in `mpirun`'s address space, that will be used for restoring the applications at restart. It then signals all the MPI processes about the pending checkpoint request by instructing the relevant `lamds` to invoke `cr_checkpoint` for every process that is a part of this MPI job. Once this is done, the callback thread indicates that `mpirun` is ready to be checkpointed.

In the MPI library, MPI_INIT has been modified to invoke the initialization function of the `blcr` checkpoint/-restart SSI module; this function registers thread-based and signal-based callbacks with BLCR, that will be executed when a checkpoint request arrives. To avoid race conditions, the current implementation defines that it is not possible to checkpoint an MPI job in which one of the processes has already completed executing MPI_FINALIZE. In order to prevent this situation from occurring, a barrier synchronization has been introduced in MPI_FINALIZE.

When a checkpoint request is received by an MPI process from `mpirun`, the threaded callback in the `blcr` module starts executing. The use of a threaded callback here allows the application to continue even when the thread-based callback starts executing. Another reason for using a threaded callback is the non-reentrancy issue mentioned above. Consequently, we have to explicitly synchronize these threads so that the application thread does not execute an MPI call when the callback thread is quiescing the network.

Synchronization of threads is already done in LAM/MPI when the thread level is MPI_THREAD_SERIALIZED, effectively preventing multiple threads from making MPI calls simultaneously. This is accomplished by placing a mutex at the entry and exit points of all MPI library calls. This same mechanism is reused in the checkpoint/restart implementation to prevent the application thread from calling into the MPI library when the callback thread is performing checkpoint or restart functions, and vice versa. Hence, all MPI applications that request checkpoint/restart support are assigned a thread level of at least MPI_THREAD_-SERIALIZED.

At checkpoint time, the callback thread in each process waits for the application thread to exit its current MPI call (if any), and then instructs the RPI to prepare itself for checkpoint. It is possible, however, that the application thread could be blocking on an MPI operation whose corresponding peer operation has not been posted. To handle this case, the callback thread of that process signals the application thread to interrupt its blocking behavior. At this point, the application thread realizes that it has been interrupted by the callback thread, and yields control to it by releasing the mutex. The callback thread can then trigger the RPI to quiesce the network, and perform any other operations that are required to prepare the process to be checkpointed. At restart time, the interrupted MPI call is automatically resumed without the user being aware of the interruption. Figures 7 and 8 depict the synchronization that is enforced between the application callback threads.

Time	CR Thread	App Thread
t_i	sleep	execute outside MPI library
t_{i+1}	wakeup	
t_{i+2}	acquire mutex	
t_{i+3}	prepare RPI for checkpoint	call MPI function, block on mutex
t_{i+4}	checkpoint	
t_{i+5}	RPI continue/restart	
t_{i+6}	release lock	
t_{i+7}	sleep	acquire lock, execute MPI call

Figure 7: Sequence of events when the application thread is executing outside the MPI library when a checkpoint request arrives.

In order to drain the in-flight data on the network, each process needs to know how much data has been sent across a TCP socket by its peer. This is accomplished by having each MPI process keep a bookmark for each of its peers. A bookmark is a pair of integers containing the number of bytes it has sent to and received from each peer.

At checkpoint time, the callback threads in each process exchange the "sent" bookmarks with each of their peers using LAM's out-of-band channel (see Figure 2). If the "sent" bookmark received from a peer does not match the "received" bookmark that the process has for that peer, then there must be some messages on the network that have not yet been received. If this is the case, the callback threads call the RPI modules to progress the receives in their internal message-passing state machines and consume data from TCP sockets until each "received" bookmark matches its corresponding "sent" bookmark. The RPI's state machine executes the normal progression of MPI receive requests by matching the posted receives with incoming messages, and creating unexpected message buffers for unmatched incoming messages. For example, if a process had posted an MPI receive before checkpoint and the message arrives after the quiesce process begins, it will be received into the actual destination buffer when the RPI drains the network. Hence, no secondary buffers or rollback mechanisms need to be utilized [10]. At this time, MPI send requests are prevented from making progress so that no more messages are

Time	CR Thread	App Thread
t_i	sleep	call MPI function, acquire mutex
t_{i+1}	wakeup	execute blocking system call in MPI library
t_{i+2}	try to acquire mutex, fail	
t_{i+3}	signal app thread	
t_{i+4}		system call interrupted, release mutex
t_{i+5}	acquire mutex	block on mutex
t_{i+6}	prepare for checkpoint	
t_{i+7}	checkpoint	
t_{i+8}	continue/restart	
t_{i+9}	release lock	
t_{i+10}	sleep	acquire lock, resume MPI function

Figure 8: Sequence of events when the application thread is executing a blocking system call inside the MPI library when a checkpoint request arrives.

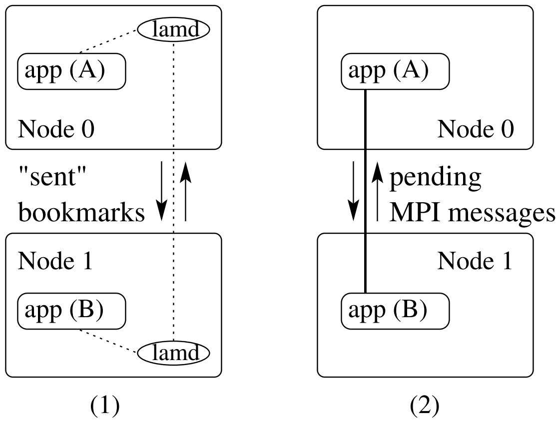

Figure 9: Clearing the communication channels before checkpoint. (1) processes A and B exchange the "sent" bookmarks that they have for each other using the out-of-band channel. (2) processes A and B receive data from the in-band channel until their "received" bookmarks match the "sent" bookmarks sent by the peer in (1).

sent. When all the bookmarks match, the RPI has drained all the in-flight data on the network, and the callback thread in each process indicates that the process is ready to be checkpointed. The underlying checkpointer then writes the process image to stable storage. Figure 9 depicts the exchange of bookmarks and the draining of in-flight data for a two-process MPI job.

4.2 Continue

After checkpoints are taken, the MPI processes are allowed to continue execution. At checkpoint time, the TCP sockets are not closed so the MPI processes need not perform additional work to re-establish connections or renegotiate per-job parameters when they continue from a checkpoint. The MPI library is unlocked and control is simply returned to the application thread and processing continues as if nothing happened.

4.3 Restart

When a checkpointed MPI job is restarted by invoking the BLCR utility `cr_restart` with the name of `mpirun`'s saved process context, the signal-based callback function `exec()`s a new `mpirun`. `mpirun` restarts all the MPI processes from the application schema that was saved at checkpoint-time, with the same process-topology as before checkpointing. A signal-based callback is required here because invoking `exec()` from another thread would result in a changed process ID on current Linux kernels (version 2.4 or earlier).

When the MPI processes resume execution, the thread-based callbacks still have the MPI library locked, with the application threads either blocked at the entry point to an MPI function, safely interrupted in their MPI function calls, or running entirely outside the MPI library. The checkpoint/restart implementation in LAM/MPI does not rely on the existence of support for transparent migration of sockets in the back-end checkpointer for performance reasons and to minimize the requirements on the underlying system. Hence, the threaded callback re-establishes new TCP sockets with each of its MPI peers. Once these connections have been re-established, the MPI library is unlocked, the callback thread completes execution, and the application thread continues.

5 Communication Performance

Experiments were conducted to measure the communication and computation performance of the checkpoint/restart system in LAM/MPI 7.0 using NetPIPE (A Network Protocol Independent Performance Evaluator) [29] and the NAS Parallel Benchmarks [11] on a Linux cluster consisting of 208 2.4 GHz Xeon processors with Fast Ethernet interconnect. NetPIPE is a program that performs ping-pong tests, bouncing messages of increasing size between two processes across a network in order to measure communication performance. The NAS Parallel Benchmarks is a suite of application kernels that test several dif-

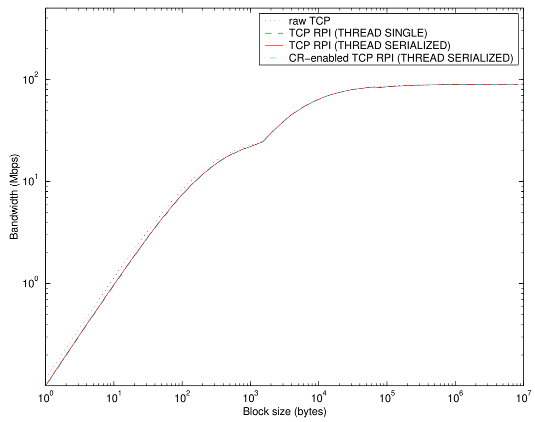

Figure 10: Performance comparison of raw TCP, plain TCP RPI (MPI_THREAD_SINGLE and MPI_THREAD_SERIALIZED) and TCP RPI with checkpoint/restart (MPI_THREAD_SERIALIZED) using NetPIPE.

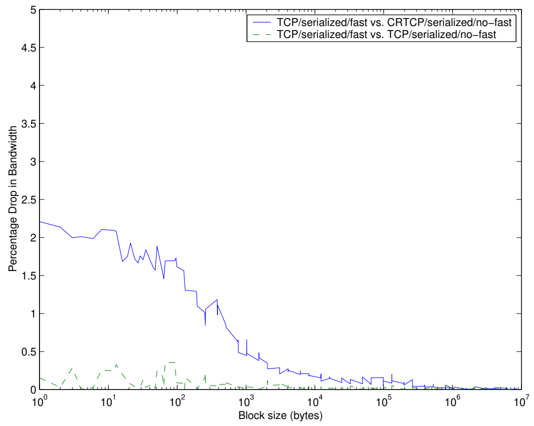

Figure 11: Performance degradation of checkpoint/restart-enabled TCP RPI (MPI_THREAD_SERIALIZED) and plain TCP RPI (MPI_THREAD_SERIALIZED) without "fast" mode, both relative to plain TCP RPI (MPI_THREAD_SERIALIZED) with "fast" mode.

ferent computational and communication patterns in parallel environments.

Experiments were conducted to measure the communications overhead of adding checkpoint/restart capability to LAM/MPI. First, the drop in performance caused by the addition of checkpoint/restart support to the TCP RPI was measured. NetPIPE was used to compare the throughput of plain TCP RPI with that of the TCP RPI with checkpoint/restart. The graph of throughput versus block-size is shown in Figure 10. The percentage of bandwidth loss in the checkpoint/restart-enabled TCP RPI as compared to plain TCP RPI is shown in Figure 11.

There are three reasons for the drop in performance of the TCP RPI with the addition of checkpoint/restart support. First, there is a "fast" mode of communication in the RPI layer such that in certain cases when the MPI request queues are empty, LAM bypasses the entire RPI state machine and directly uses sends and receives for performance reasons. The current implementation of the checkpoint/restart enabled TCP RPI does not support this "fast" mode of communication, and based on running tests with the "fast" mode disabled in the TCP RPI, it has been determined that this accounts for a part of the deterioration in performance that is seen in the graphs (see Figure 11. Second, when an MPI job requests checkpoint/restart support, the thread level is automatically upgraded to MPI_THREAD_SERIALIZED. In this situation, LAM uses mutexes to synchronize the threads, and this leads to additional overhead due to the lock/unlock operations that need to be performed every time an MPI call is made. A third reason for the degradation in performance is the additional book keeping that is done in the RPI layer to support checkpoint/restart. Since checkpoint/restart functionality adds a constant overhead to the MPI layer, performance drop is maximum for small sized messages (about 2 percent). For messages larger than 1KB, the performance degradation is less than 0.5 percent.

To assess the impact of LAM's checkpoint/restart infrastructure on the computational performance of parallel applications, the entire suite of NAS Parallel Benchmarks (problem size class A) were run on four nodes both with and without checkpoint/restart support. There was no discernable difference in the wall-clock execution time of any of the benchmark applications.

6 Future Work

Future work on this project is planned in several directions. Our first priority for future work is to implement the "fast" mode of communication in the modified TCP RPI and extend checkpoint/restart support to all the the remaining RPIs so that it will be possible to checkpoint/restart all MPI-1 jobs running in LAM/MPI. The next step will be to extend the implementation to include MPI-2 functionality. Later, we plan to look into the possibility of building checkpoint/restart SSI modules on top of other back-end checkpointing systems, possibly including Condor [23], Libckpt [25] and CRAK [41], to extend our implementation to multiple platforms. All of these efforts will be complemented with extensive performance testing and tuning to understand and identify run-time bottlenecks. Another possibility for future work in this project is full support for process migration. Our current implementation lets us restore an entire checkpointed job on a different set of nodes in some cases, but it does not permit us to migrate a subset of the processes while the others are still running. While support for "real-time" migration would be contingent upon the underlying system's ability to do this, additional work also needs to be done in the MPI library itself to make this possible. Finally, a long term goal is to investigate the implementation of an uncoordinated approach to checkpointing MPI jobs in LAM/MPI.

7 Conclusions

This paper presented a checkpoint/restart implementation for MPI jobs that has been implemented in LAM/MPI using BLCR [8] as the underlying checkpointer. This implementation adopts a coordinated approach to checkpointing jobs. The performance of this system was tested to measure the overhead of adding checkpoint/restart functionality, and the time to checkpoint MPI jobs. Experiments have shown that the drop in performance caused by the introduction of additional functionality in the MPI layer and the communication sub-system is negligible, and the time to checkpoint jobs increases linearly with the number of processes.

The checkpoint/restart system and all other modifications to the LAM infrastructure that grew out of this project are currently available in LAM's CVS tree. Anonymous read-only access is available to users who wish to utilize the latest features in LAM/MPI. The checkpoint/restart functionality is also scheduled to be included in the upcoming LAM/MPI 7.0 release. More information on the project can be found on the web:

`http://www.lam-mpi.org/`

Acknowledgments

This work was supported by a grant from the Lilly Endowment, by National Science Foundation grant 0116050, and by the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. Brian Barrett was supported by a Department of Energy High Performance Computer Science fellowship.

References

[1] Top500 supercomputer list, November 2002. http://www.top500.org/.

[2] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *Proceedings of the Fourth International Symposium on Reliability in Distributed Software and Databases*, pages 207–215, 1984.

[3] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In J. W. Ross, editor, *Proceedings of Supercomputing Symposium '94*, pages 379–386. University of Toronto, 1994.

[4] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computing Systems*, 3(1):63–75, 1985.

[5] Y. Z. Chang, K. S. Ding, and J. J. Tsay. Efficient Implementation of Message Passing Interface on Local Area Networks, 1996.

[6] Y. Chen, K. Li, and J. S. Plank. CLIP: A checkpointing tool for message-passing parallel programs. In ACM, editor, *SC'97: High Performance Networking and Computing: Proceedings of the 1997 ACM/IEEE SC97 Conference: November 15–21, 1997, San Jose, California, USA.*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. ACM Press and IEEE Computer Society Press.

[7] W. R. Dieter and J. E. L. Jr. A user-level checkpointing library for POSIX threads programs. In *Symposium on Fault-Tolerant Computing*, pages 224–227, 1999.

[8] J. Duell, P. Hargrove, and E. Roman. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart, 2002.

[9] E. N. Elnozahy, D. B. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, Oct. 1992.

[10] 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, 1996.

[11] D. H. B. et al. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Moffett Field, CA, 1994.

[12] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the Message-Passing Interface. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par '96 Parallel Processing*, number 1123 in Lecture Notes in Computer Science, pages 128–135. Springer Verlag, 1996.

[13] Genias Software GmbH. *CODINE User's Guide*, 1993.

[14] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI — The Complete Reference: Volume 2, the MPI-2 Extensions*. MIT Press, 1998.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[16] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, 1994.

[17] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, 1999.

[18] W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[19] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. Technical Report TR85-706, Cornell University, Computer Science Department, 1985.

[20] W.-J. Li and J.-J. Tsay. Checkpointing Message-Passing Interface (MPI) Parallel Programs. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, 1997.

[21] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, pages 104–111, 1988.

[22] M. Litzkow and M. Solomon. The Evolution of Condor Checkpointing, 1998.

[23] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System. Technical Report CS-TR-1997-1346, University of Wisconsin, Madison, Apr. 1997.

[24] Message Passing Interface Forum. MPI: A Message Passing Interface. In *Proc. of Supercomputing '93*, pages 878–883. IEEE Computer Society Press, November 1993.

[25] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent Checkpointing under Unix. In *Proceedings of the 1995 Winter USENIX Technical Conference*, 1995.

[26] B. Randell. Systems structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[27] D. L. Russell. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, 6(2):183–194, Mar. 1980.

[28] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine. Checkpoint-restart support system services interface (SSI) modules for LAM/MPI. Technical Report TR578, Indiana University, Computer Science Department, 2003.

[29] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. NetPIPE: A Network Protocol Independent Performace Evaluator, 1996.

[30] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, 1996.

[31] J. M. Squyres, B. Barrett, and A. Lumsdaine. Request progression interface (RPI) system services interface (SSI) modules for LAM/MPI. Technical Report TR579, Indiana University, Computer Science Department, 2003.

[32] J. M. Squyres, B. Barrett, and A. Lumsdaine. The system services interface (SSI) to LAM/MPI. Technical Report TR575, Indiana University, Computer Science Department, 2003.

[33] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, Euro PVM/MPI*, October 2003.

[34] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, Honolulu, HI, 1996.

[35] Y. Tamir and C. H. Sequin. Error recovery in multicomputers using global checkpoints. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 32–41, Bellaire, Michigan, Aug. 1984. IEEE.

[36] The BlueGene/L Team. An Overview of the BlueGene/L Supercomputer, 2002.

[37] The LAM Team. *Getting Started with LAM/MPI*. University of Notre Dame, Department of Computer Science, `http://www.lam-mpi.org/`, 1998.

[38] Z. Tong, R. Y. Kain, and W. T. Tsai. Rollback recovery in distributed systems using loosely synchronized clocks. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):246–251, 1992.

[39] Y.-M. Wang, P.-Y. Chung, I.-J. Lin, and W. K. Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546–554, 1995.

[40] A. Wong, L. Oliker, W. Kramer, T. Kaltz, and D. Bailey. System Utilization Benchmark on the Cray T3E and IBM SP, April 2000.

[41] H. Zhong and J. Nieh. CRAK: Linux checkpoint / restart as a kernel module. Technical Report CUCS-014-01, Department of Computer Science, Columbia University, 2001.