The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI *

Joshua Hursey[1], Jeffrey M. Squyres[2], Timothy I. Mattox[1], Andrew Lumsdaine[1]

[1]Indiana University
Open Systems Laboratory
Bloomington, IN USA
{jjhursey, timattox, lums}@osl.iu.edu

[2]Cisco Systems, Inc.
Server Virtualization Business Unit
San Jose, CA USA
jsquyres@cisco.com

Abstract

To be able to fully exploit ever larger computing platforms, modern HPC applications and system software must be able to tolerate inevitable faults. Historically, MPI implementations that incorporated fault tolerance capabilities have been limited by lack of modularity, scalability and usability. This paper presents the design and implementation of an infrastructure to support checkpoint/restart fault tolerance in the Open MPI project. We identify the general capabilities required for distributed checkpoint/restart and realize these capabilities as extensible frameworks within Open MPI's modular component architecture. Our design features an abstract interface for providing and accessing fault tolerance services without sacrificing performance, robustness, or flexibility. Although our implementation includes support for some initial checkpoint/restart mechanisms, the framework is meant to be extensible and to encourage experimentation of alternative techniques within a production quality MPI implementation.

1. Introduction

High Performance Computing (HPC) systems continue to grow in both complexity and size. As these systems grow they suffer from an increased opportunity for various system failures. Many of these systems do not provide transparent fault tolerance services at the system level. Modern HPC applications generally rely on message passing libraries such as the Message Passing Interface (MPI) for inter-process communication [14]. MPI is positioned such that it has unique knowledge about the state of the parallel job and available resources making it a natural place for incorporating fault tolerance techniques.

Many MPI implementations couple fault tolerant techniques tightly throughout their code. This makes it difficult for researchers to explore new techniques and refinements without forking from the base MPI implementation. These fault tolerant MPI libraries assume that MPI is the only service that the application depends upon which needs to be aware of a checkpoint or restart request. Many of the fault tolerance interfaces and tool sets are cumbersome which deter users from exploiting these features.

This paper addresses these issues in a new design incorporated into the Open MPI project [8]. This paper first distills the primary duties involved in a distributed checkpoint and restart system into five primary tasks. From these tasks this paper presents a modular design and implementation which isolates each task. This modular isolation empowers researchers to easily explore new techniques and refinements. The modular design also allows for multiple implementations of a task to be interchangeable at runtime. This paper will discuss some of the implementation details involved with incorporating this design into Open MPI.

An application using a MPI implementation that provides fault tolerance services is better equipped to adapt to current and next generation HPC systems. Unfortunately many of these implementations provide complicated interfaces to the application user or require them to decide between production quality and research quality implementations.

There are many fault tolerance techniques available. Checkpoint and restart techniques are a member of the rollback recovery class of fault tolerance techniques, and have been widely adopted by many fault tolerance implementations. These techniques capture an image (or *snapshot*) of a running process and preserve it for later recovery. Distributed checkpoint and restart techniques rely on various coordination protocols to produce consistently recoverable

*Supported by a grant from the Lilly Endowment and National Science Foundation grants NSF ANI-0330620, CDA-0116050, and EIA-0202048.

parallel application states.

Open MPI is a high performance, production quality, MPI-2 standard compliant implementation. The design presented in this paper augments this implementation providing users with the option of using checkpoint/restart fault tolerance techniques.

This design handles synchronous and asynchronous checkpoint requests. Synchronous checkpoint requests are handled by an application via a common API. Asynchronous checkpoint requests are handled by various command line tools. These tools enable system administrators and support services (e.g., schedulers) the ability to checkpoint a user's job for various reasons such as system maintenance. In order for these tools and interfaces to be widely used they must provide a simple yet robust interface that makes it easy and flexible to use in production environments.

The end goal of the fault tolerance implementations in Open MPI is to provide a high performance, scalable, transparent MPI fault tolerance solution. This paper presents the first step in that direction for Open MPI. Extensions of this design include the exploration of message logging[2], migration[15], automatic recovery, and gang scheduling.

2. Related Work

A checkpoint/restart system is responsible for saving the current state of a single process for later restart. Many checkpoint/restart system implementations are available such as: libckpt [16], Condor checkpoint library [12], and BLCR (Berkley Lab's Checkpoint/Restart) [5]. These implementations differ in many ways including how process state is preserved, how much of the process state is preserved, how the state is stored, APIs, and command line interfaces.

Distributed checkpointing and restarting of parallel applications requires the coordination between individual processes to create consistent recoverable states for the parallel application. There have been many techniques proposed in literature [6], with most techniques falling into one of three categories: coordinated, uncoordinated, and communication (or message) induced.

A few MPI libraries have attempted to integrate fault tolerance techniques. The techniques integrated range from user interactive process fault tolerance (FT-MPI [7]) to network failure recovery (LA-MPI [10]). Other MPI implementations integrate checkpoint/restart techniques to save and restore the state of the parallel application. Starfish [1] provides support for coordinated and uncoordinated checkpoint/restart protocols. Egida [18] provides a grammar for experimenting with new message logging protocols. CoCheck [20] uses the Condor [12] checkpoint/restart system and a ready message checkpoint/restart coordination protocol. That protocol allowed a subset of the processes in the job to quiesce their network channels before taking a checkpoint.

MPICH-V [3] uses an uncoordinated checkpoint/restart protocol in conjunction with message logging to preserve the process state and automatically recover failed processes. This project introduces the use of event logger and checkpoint server processes which assist in alleviating the overhead of these fault tolerant techniques. They use the Condor [12] checkpoint/restart system.

Many of these implementations are tightly coupled with a specific checkpoint/restart system. LAM/MPI modularized its checkpoint/restart approach and allowed support for integrating multiple checkpoint/restart systems into its code base [19]. They support communication over both TCP and Myrinet interconnects. They support the BLCR and *SELF* checkpoint/restart systems, the latter being a set of user-level callbacks to assist in user-level checkpointing. The framework design allowed LAM/MPI to incorporate various checkpoint/restart systems uniformly throughout.

LAM/MPI only supports a coordinated checkpoint/restart protocol, and therefore only supports the checkpoint and restart of the entire parallel application. LAM/MPI also requires that the checkpoint/restart systems provide a notification thread to `mpirun` to initiate the checkpoint of a parallel job. The command line tools provided for asynchronous checkpointing require the user to recall the exact parameters that were used to launch the original process in order to checkpoint and restart it correctly. LAM/MPI did not provide any API for synchronous checkpointing and/or restarting from within a process.

The design presented by this paper extracts the best practices from previous implementations, and incorporates them with some refinements into Open MPI. The modular design presents researchers with a medium for the accurate comparison of various fault tolerance techniques keeping all other variables constant. Usability improvements in tool parameters and APIs encourage end users to use the fault tolerance features presented. One such API addition is the notification callback surrounding checkpoint and restart requests allowing modern HPC applications the opportunity to maintain libraries and services outside of MPI.

3. Open MPI Architecture

Open MPI consists of three abstraction layers that combine to provide a full featured MPI implementation. Below the user application is the *Open MPI* (OMPI) layer that presents the application with the expected MPI standard interface. Below that is the *Open Run-Time Environment* (ORTE) layer that provides a uniform parallel run-time interface regardless of system capabilities. Next is the *Open Portable Access Layer* (OPAL) that abstracts the peculiarities of a specific system away to provide a consistent interface aiding portability. Below OPAL is the operating system and other standard services running on the local machine.

Open MPI uses the Modular Component Architecture

(MCA) to define internal APIs called *Frameworks* for particular services such as process launch. Each framework contains one or more *Components* which are specific implementations for a framework (e.g., SLURM and RSH components of the process launch framework). Components can be dynamically selected at runtime.

3.1. MPI Standard Coverage

The implementation of the design presented in this paper ultimately desires to support the entire MPI-2 standard [9]. However there exist many implementation and MPI standard functionality support issues that make this difficult, for example: how to properly handle dynamic processes, one-sided communication and hardware collectives.

Many fault tolerance MPI implementations only support the MPI-1 standard [14] when using these techniques. The MPI-1 standard tends to be sufficient for many MPI applications. This first implementation provides a solid foundation of MPI-1 support, and support for MPI collective routines when internally layered over point-to-point communication. This foundation was designed to be built upon in the future to support additional portions of the MPI standards, and component advancements (e.g., hardware collectives).

4. Snapshot Representations

One of the hurdles standing in the path of wider user fault tolerance adoption is usability and integration overhead. The fault tolerance interfaces provided by the system should be convenient, intuitive and easy to use without sacrificing robustness and flexibility.

Previous implementations required the user to remember specifics regarding exactly how the job was started in order to successfully checkpoint and/or restart the parallel job. Since long running applications are the primary target for fault tolerance acceptance it is common that the user will not remember all the runtime specifics at a later time. Further a system administrator or scheduler developer is completely precluded from this runtime information, and must consult the user before taking action with a job.

Additionally, these same implementations tended to burden the user with the responsibility of tracking the location of all the checkpoint generated files. Depending on the checkpoint/restart system the file set may be one or many files with specific naming conventions. Tracking sets of requests quickly becomes tedious and error prone.

The design presented in this paper addresses both of these issues by introducing an abstract *snapshot reference*. A snapshot reference is a single named reference to the checkpoint that was taken of a single process or a parallel job. There are two types of snapshot references. The *local snapshot reference* is a single process checkpoint. The local snapshot reference refers to a directory containing a metadata file describing: the checkpointer used, application specific parameters, and checkpoint interval information. The directory also contains all of the single process checkpoint/restart system specific files. Each snapshot generated is designated an *interval* number that differentiates one from another in a logical ordering.

The second type of snapshot reference is the *global snapshot reference* that references a collection of local snapshots resulting from a single checkpoint request. The global snapshot reference is represented as a directory containing a metadata file describing: the aggregated local snapshot references, process information (e.g., last known rank), runtime parameters, and a global checkpoint interval. The global snapshot directory also contains the physical set of local snapshots, one from each process in the checkpoint interval.

The snapshot references abstract the user away from the number and name of the checkpoint generated files alleviating the need for them to track multiple different files for a single distributed checkpoint interval. The user is only responsible for the preservation of a directory containing all the relevant checkpoint information. Additionally the user does not need to know the underlying checkpoint/restart system used in order to properly preserve the checkpoint files.

This design alleviates the need for the user to know which runtime parameters the job was originally started with by automatically detecting them when checkpointing and placing a reference to the parameters in the metadata files in the snapshot references. During restart the metadata files are used to determine how to restart the entire job properly.

This level of abstraction allows for the possibility of heterogeneous checkpoint/restart system support. Single process checkpoint/restart systems tend to be closely tied to the operating system on which they run, and generate binary files intended to be restarted on the same type of system. A job spanning a heterogeneous environment must incorporate the checkpoints produced by potentially different checkpoint/restart systems into a single global snapshot. The files generated from these distinct checkpoint/restart systems are likely to be incompatible due to implementation differences, but can still be incorporated into the same global snapshot if the restart mechanism is able to properly map onto the heterogeneous environment as required by the global snapshot.

Asynchronous checkpointing is supported by a variety of command line tools. They allow the user to checkpoint/restart processes, and disconnect from a long running checkpoint operation.

5. Design

Five primary tasks were distilled from a review of previous fault tolerant MPI implementations that incorporate distributed checkpoint and restart techniques.

- Launching, monitoring and aggregation of checkpoint requests.

- Managing checkpoint related files and directories in a distributed environment incorporating potentially local and global file systems.
- Incorporating the checkpoint/restart coordination protocol that guarantees a consistently recoverable distributed state upon restart [4].
- Interfacing with a single process checkpoint/restart system provided by or for the system.
- Notifying and coordinating subsystems of the MPI implementation around a checkpoint or restart request.

Previous fault tolerant MPI implementations incorporating checkpoint and restart fault tolerance techniques have tightly integrated some subset of these tasks with the MPI implementation. This tight integration makes it difficult to explore alternative techniques.

5.1. Snapshot Coordinator

Distributed checkpoint/restart implementations are all required to do the following tasks upon receiving distributed checkpoint/restart requests: initiate the per process local checkpoints; monitor the progress of the global checkpoint; aggregate the local checkpoints into a global checkpoint and preserve it on a stable storage medium.

Implementations of this distributed checkpoint/restart task should be given the flexibility to support a wide variety of snapshot coordination techniques. Example techniques include the spawning of replicated checkpoint servers, initiating multiple local checkpoints concurrently in a hierarchal tree structure, and grouping remote file movement request as to avoid network congestion.

Processes should have the ability to choose between being able to be checkpointed and not. Processes may choose not to be checkpointable for various reasons including the use of unsupported algorithms such as hardware collectives or dynamic operations. The snapshot coordinator is responsible for taking the checkpoint request from the user and checking this against the processes that have identified themselves as able to be checkpointed. If any of the processes in the checkpoint request cannot be checkpointed then the user should be notified and no processes participating in the request should be affected.

5.2. File Management

Remote file management enables the runtime system to preload files or binaries on remote systems before starting remote processes providing usability conveniences.

This task must support broadcast, gather, and remove operations. The broadcast operation supports the preloading of checkpoint related files on remote machines during process recovery. The gather operation supports the movement of remote local snapshots to a stable storage medium. The remove operation allows for cleanup of temporary checkpoint data that was preloaded on a remote machine.

A stable storage system is defined as a storage medium that ensures that the recovery information persists through the tolerated failures and their corresponding recoveries [6]. In practice, non-transient failure of one or more machines in the system needs to be tolerated. Therefore many administrators provide a shared RAID file system that persists past the failure of any machine in the system.

Many methods exist for physically moving a file from a local to a potentially remote file system including standard UNIX and RSH copy commands. The interface should allow multiple file management requests to be given to the file management system at the same time. This interface allows it to use collective algorithms to optimize the operation.

5.3. Distributed Checkpoint/Restart Coordination Protocol

A snapshot of a process is defined as the state of the process and all connected communication channels [4]. Local checkpoint/restart systems are unable to account for the state of communication channels as they require knowledge of and the ability to coordinate with remote processes. Provided this restriction a higher level protocol is required to coordinate all the processes to create known channel states. Knowing the state of all connected communication channels is critical when forming a consistent global snapshot of the parallel job from which the process can be accurately restarted at a later time. Many checkpoint/restart coordination protocols exist and can be generally classified into one of three categories: coordinated, uncoordinated, and communication or message induced [6]. Each protocol balances the demand for low overhead failure-free operation with the complexity of recovery in the event of unexpected process termination due to system failure.

Distributed checkpoint/restart coordination protocol services must provide a consistent API for the MPI implementation to use internally when such a protocol is required. Many protocols require the ability to track all point-to-point messages in the system to aid in recovery [21]. Other protocols require the ability to piggyback data on outgoing messages, and take action on incoming messages such as taking forced checkpoints [13]. Therefore these coordination services need to be provided access to the MPI implementations internal point-to-point layer. By doing so these coordination services are then allowed to watch the network traffic as it moves through the system and take necessary actions.

Coordination services should receive checkpoint notification before any MPI subsystem. This ordering provides coordination services flexibility in their protocol implementation by not restricting the MPI subsystems available.

5.4. Local Checkpoint/Restart System

Many single process checkpoint/restart systems exist for various platforms. Examples include BLCR [5], libckpt [16], and Condor [12]. System level checkpoint/restart sys-

tem implementations tend to be tied to a specific operating system type or revision. This tight coupling provides them with a more detailed view of the process target allowing for a more detailed coverage of the process in the checkpoint. User level implementations tend to exist above the operating system making them more portable by sacrificing their ability to view some process details. Both varieties of checkpoint/restart systems capture a snapshot of a single process on the system and save it to storage. Many times they are not able to account for the state of entities that exist outside of the process scope such as file system or network interconnect states which maybe required for the proper recovery of an application.

In essence a local checkpoint/restart system is required to provide the following two tasks:

- Request a checkpoint of a specific PID, and return a reference to the generated local snapshot for later restart. The PID may be that of the requesting process or another process on the same machine.
- Request the restart of a process on the local machine provided a local snapshot reference generated by the checkpoint command.

Additional functionality may be added to the design such as memory inclusion and exclusion hints for checkpoint/restart systems that support such operations [17].

5.5. MPI Library Notification Mechanisms

A single process checkpoint/restart system may only preserve a subset of the process state. As previously mentioned, they tend not to account for the state of communication channels. Therefore subsystems within an MPI implementation need to receive notification around checkpoint/restart requests.

In this design each subsystem that requires such a notification implements a `ft_event` function defined as follows: `int ft_event(int state);`. This function is meant to encapsulate most, if not all, of the subsystem specific logic needed to respond to a checkpoint and/or restart notification. By attempting to isolate this logic to this function checkpoint/restart notifications can have a minimal impact upon the implementation of the subsystem making the entire checkpoint/restart integration more maintainable. The `ft_event` function takes a single state argument indicating the state of the checkpoint/restart protocol at the time of the function call.

A driver notification routine is responsible for calling each subsection's `ft_event` function in the proper order upon receiving a checkpoint or restart request. These routines are called *Interlayer Notification Callbacks* (INCs). For a monolithic library design only a single INC may be needed. For library designs involving multiple layers of abstraction, such as Open MPI, one INC may be needed for each layer. Once the INCs finish preparing the library for a checkpoint, it then calls the single process checkpoint/restart system. Once the checkpoint has completed then it uses the `ft_event` function to notify the subsystems of the resulting state of the process.

The presented design provides the MPI library the opportunity to prepare for and respond to checkpoint/restart requests. However since some modern HPC applications may also need to be notified of such requests the design needs to be extended to provide such a notification. The application can be viewed as a layer existing above the MPI library. Therefore the design can present the application with functionality to register an INC. Multilayered MPI libraries can use this mechanism to register their INCs. INCs have the same function definition as `ft_event`. The INCs are registered by calling a registration function which will return the previous registered callback. It is the newly registered INC's responsibility to call the previous INC from within their own. This responsibility ensures a stack-like ordering of INC calls, and gives an INC the opportunity to take action before and after calling the previous INC which is often from a lower level in the software hierarchy. An example of this is described in Section 6.5.

6. Implementation

This section explores the implementation in Open MPI of the five primary tasks identified in Section 5. Open MPI's modular architecture allows for each of the five tasks to be logically separated into frameworks. Components of these frameworks present different techniques for achieving the goals of the individual frameworks.

The components implemented in this first round are reference implementations intended to prove the completeness of the design. The implementation presented provides Open MPI with a LAM/MPI-like checkpoint/restart implementation.

6.1. Snapshot Coordinator

There exist many different techniques for achieving the duties described in Section 5.1. Open MPI provides the ORTE SNAPC framework to compartmentalize these techniques into components with a common API. This compartmentalization allows for a side-by-side comparison of these techniques in a constant environment.

The initial ORTE SNAPC component implements a centralized coordination approach. It involves three subcoordinators: a *global coordinator*, a set of *local coordinators* and a set of *application coordinators*. Each subcoordinator is positioned differently in the runtime environment as shown in Figure 1.

The global coordinator is a part of the `mpirun` process. It is responsible for interacting with the command line tools (Figure 1-A), generating the global snapshot reference, aggregation of the remote files into a global snapshot (Fig-

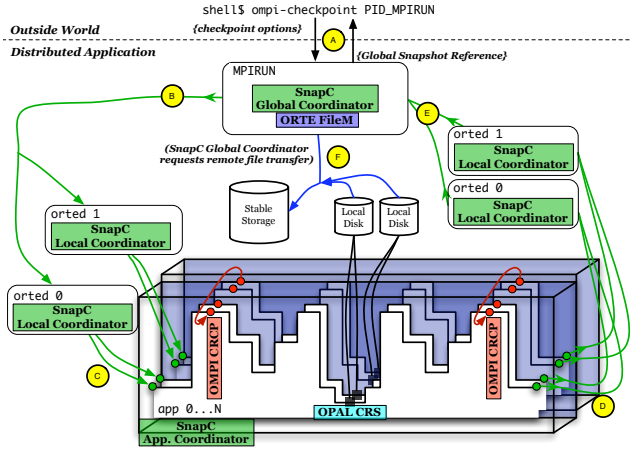

Figure 1. Illustration of Open MPI frameworks participating in a distributed checkpoint. 3D boxes represent nodes containing white application processes. Rounded boxes represent processes.

ure 1-F), and monitoring the progress of the entire checkpoint request (Figures 1-B,E).

The local coordinator is a part of the ORTE per node daemons (orted). It works with the global coordinator to initiate the checkpoint of a single process on their machine (Figure 1-C), and to move the files back to the global coordinator for storage as a part of the global snapshot (Figure 1-F).

The application coordinator is a part of each application process in the distributed system. This coordinator is responsible for starting the single process checkpoint. Such a responsibility involves interpreting any parameters that have been passed down from the user (e.g., checkpoint and terminate), and calling the OPAL `entry_point` function which begins the interlayer coordination mechanism shown in Figure 2.

Once the application coordinator has completed the process checkpoint it notifies the local coordinator (Figure 1-D) that in turn notifies the global coordinator (Figure 1-E). The global coordinator then requests the transfer of the local snapshots while the processes resume normal operation (Figure 1-F). Once these local snapshots have been aggregated and saved to stable storage the global snapshot reference is returned to the user (Figure 1-A).

6.2. File Management

Open MPI provides a file management framework entitled ORTE FILEM. This implementation requires knowledge of all of the machines in the job, but does not require knowledge of MPI semantics therefore it is implemented as a part of the ORTE layer. The framework interface provides Open MPI the ability to pass a list of peers and local and remote file names. If the remote file location is unknown by the requesting process then the remote process is queried for its location.

The first component available for the ORTE FILEM framework uses RSH/SSH remote execution and copy commands. Additional components of this framework may include standard UNIX commands and high performance out-of-band communication channels.

6.3. Distributed Checkpoint/Restart Coordination Protocol

Open MPI provides a checkpoint/restart coordination protocol framework entitled the OMPI CRCP (Checkpoint/Restart Coordination Protocol). Since this framework may require knowledge of MPI semantics it is placed at the OMPI layer. The OMPI CRCP framework provides a consistent API for Open MPI to use internally when such a protocol is required. Each component implements a single protocol. The components are provided access to the internal point-to-point management layer framework (PML) [22] by way of a wrapper PML component. The wrapper PML component allows the OMPI CRCP components the opportunity to take action before and after each message is processed by the actual PML component.

The MCA framework provides runtime selectable components allowing researchers to easily compare different protocols to each other while keeping all other variables constant. This runtime comparison produces a reproducible, accurate comparison of two proposed protocols or refinements of a given protocol. By isolating the checkpoint/restart coordination protocol to a framework, researchers can focus on the development of a small component instead of the development of an entire MPI library implementation in order to try out a new technique.

The first component of the OMPI CRCP framework is a LAM/MPI-like coordinated checkpoint/restart protocol presented in [19]. The protocol uses a bookmark exchange to coordinate processes to form a consistent global snapshot of the parallel job. This component refines the protocol slightly by operating on entire messages instead of bytes. Outstanding messages are posted by the receiving peer and used during failure free operation.

Once the OMPI CRCP component has completed its coordination of the processes then the PML's `ft_event` function is called. The PML `ft_event` function involves shutting down interconnect libraries that cannot be checkpointed and reconnecting peers when restarting in new process topologies.

6.4. Local Checkpoint/Restart System

Open MPI provides a single process checkpoint/restart service framework entitled the OPAL CRS (Checkpoint/Restart Service). Since this framework's functionality is limited to a single machine by the single process checkpoint/restart service it is implemented at the OPAL layer. The OPAL CRS framework provides a consistent API for

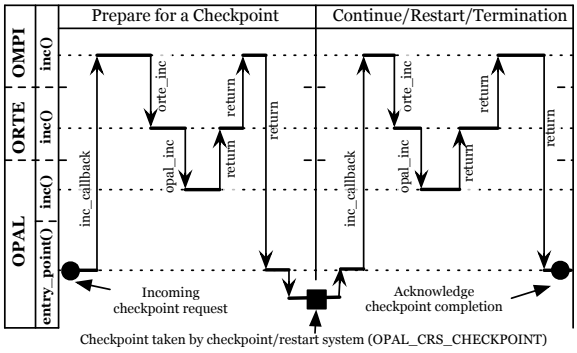

Figure 2. Illustration of Open MPI Handling a Checkpoint Request

Open MPI to use internally regardless of underlying checkpoint/restart system available on a specific machine. Each such system implements a component in the OPAL CRS framework that interfaces the framework API to the checkpoint/restart system's API.

The framework API provides the two basic operations of checkpoint and restart. In addition the OPAL CRS framework requires components to implement the ability to enable and disable checkpointing in the system to assist in protecting non-checkpointable sections of code.

In Open MPI checkpointing is enabled upon completion of MPI_INIT and disabled upon entry into MPI_FINALIZE. This restriction allows checkpointing only while MPI is enabled since the checkpoint/restart framework is a part of the MPI infrastructure and is therefore initialized and finalized within the library. The framework interface is described in more detail in [11].

There currently exist two components of the OPAL CRS framework. The first is a BLCR implementation. The second is a *SELF* component supporting application level checkpointing by providing the application callbacks upon checkpoint, restart and continue operations.

Future OPAL CRS framework API refinements will allow for checkpoint/restart system hints such as memory inclusion and exclusion operations.

6.5. Open MPI Notification Mechanisms

In Open MPI each process in the parallel job has a thread running in it waiting for the checkpoint request. This thread is called the *checkpoint notification thread*. The thread receives a checkpoint request notification from the system and proceeds into the OPAL `entry_point` function to begin the notification process, as seen in Figure 2. This function then calls the top most registered interlayer notification callback (INC) function. There are three INC functions in Open MPI, one for each layer in the software stack described in Section 3. If the application registered an INC then it has the opportunity to use the full suite of MPI functionality before allowing the library to prepare for a checkpoint.

Since the checkpoint notification thread executes concurrently with other threads in the process, the notification process typically does not interfere with the progress of the process. A thread in the process is only stopped when it tries to access a part of the Open MPI library that has been notified and restricts that particular operation from continuing until the checkpoint is complete. For example the point-to-point layer may not allow a call to MPI_SEND to begin between when a checkpoint was requested and its completion.

In Open MPI each INC uses the `ft_event` function to notify framework components of the checkpoint request. This function is an extension to existing framework APIs. Using a separate function for this type of notification has proven useful in isolating fault tolerance specific logic.

7. Results

The purpose of the infrastructure developed in this paper is to support fault tolerance in MPI applications, not to be a complete fault tolerance system. Accordingly, one important performance measurement for our design is the amount of overhead it introduces to MPI communication operations. NetPIPE latency comparison showed that Open MPI incurs about 3% overhead for small messages (0% for large messages) when using this infrastructure and passthrough components. The overhead is attributed to function call overhead. Bandwidth overhead was 0%.

Tests were run on a Linux cluster of dual 2.0 GHz Opteron nodes, each with 4 GB of RAM. Machines are connected via gigabit ethernet and Infiniband.

Regarding usability and modularity of the infrastructure, we note that once the infrastructure was in place that it took only a few weeks to fully implement the LAM/MPI-like coordinated checkpoint/restart protocol component. By way of contrast, many months were required to implement the original checkpoint/restart support directly into LAM/MPI.

8. Conclusions

HPC systems will (and do) require application programmers to adapt to adverse runtime conditions that may preclude correct parallel applications from running to completion. MPI implementations provide a single process within a job with the ability to view and interact with the entirety of a parallel job. Given MPI implementation's unique knowledge about the state of the processes and communication channels in the parallel job it is a natural candidate to assist in application fault tolerance integration. There are many different fault tolerant techniques available; checkpoint/restart rollback recovery techniques being one popular choice.

Open MPI is an open source, high performance, MPI-2 compliant implementation of the MPI standard. This paper describes the current design and implementation details for the integration of checkpoint/restart fault tolerance into Open MPI. The design presented in this paper has taken the best practices and experiences from previous projects

and presented a modular, extensible framework set. This framework set maps directly to the basic responsibilities of a distributed checkpoint/restart system. Each framework is designed to allow fault tolerance researchers sufficient flexibility in their implementations while keeping a logical separation from the rest of the code. The modular design empowers researchers by allowing them to focus on the specific algorithm designs and optimizations that interest them without the burden of supporting a production quality MPI implementation.

One of the primary hurdles for fault tolerance acceptance by the user community is ease of use. This paper identified some primary problems with previous designs, and presented solutions that are integrated into the Open MPI implementation.

Additional fault tolerance capabilities that we intend to support with our design include process migration; gang scheduling; automatic, transparent recovery; and message logging. The extensible and modular infrastructure presented in this paper will allow researchers to investigate these (and other) capabilities in the context of a production-quality MPI implementation.

References

[1] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. In *HPDC '99: Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, page 31, Washington, DC, USA, 1999. IEEE Computer Society.

[2] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Trans. Softw. Eng.*, 24(2):149–159, 1998.

[3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[5] J. Duell, P. Hargrove, and E. Roman. The design and implementation of Berkeley Lab's linux checkpoint/restart. Technical Report LBNL-54941, Lawrence Berkeley National Lab, 2003.

[6] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[7] G. E. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Lecture Notes in Computer Science: Proceedings of EuroPVM-MPI 2000*, pages 346–353, 2000.

[8] E. Garbriel, G. E. Fagg, G. Bosilica, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

[9] 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 *Euro-Par '96 Parallel Processing*, pages 128–135. Springer Verlag, 1996.

[10] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. Risinger, and M. W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *International Journal of Parallel Programming*, volume 31, pages 285–303, August 2003.

[11] J. Hursey, J. M. Squyres, and A. Lumsdaine. A checkpoint and restart service specification for Open MPI. Technical Report TR635, Indiana University, Bloomington, Indiana, USA, July 2006.

[12] 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-199701346, University of Wisconsin, Madison, 1997.

[13] D. Manivannan, R. H. B. Netzer, and M. Singhal. Finding consistent global checkpoints in a distributed computation. *IEEE Trans. Parallel Distrib. Syst.*, 8(6):623–627, 1997.

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

[15] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. Process migration. *ACM Comput. Surv.*, 32(3):241–299, 2000.

[16] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. Technical report, Knoxville, TN, USA, 1994.

[17] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. In *Software – Practice and Experience*, volume 29, pages 125–142, 1999.

[18] S. Rao, L. Alvisi, and H. M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 48, Washington, DC, USA, 1999. IEEE Computer Society.

[19] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[20] G. Stellner. CoCheck: Checkpointing and process migration for MPI,. In *10th International Parallel Processing Symposium (IPPS '96)*, page 526, 2996.

[21] Y.-M. Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Trans. Comput.*, 46(4):456–468, 1997.

[22] T. S. Woodall, G. M. Shipman, G. Bosilca, R. L. Graham, and A. B. Maccabe. High performance RDMA protocols in HPC. In *Proceedings of EuroPVM-MPI 2006*, volume 4192/2006 of *Lecture Notes in Computer Science*, pages 76–85. Springer berlin /Heidelberg, September 2006.