A Checkpoint and Restart Service Specification for Open MPI

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

Open Systems Laboratory, Indiana University
{jjhursey, lums}@osl.iu.edu

Cisco Systems, Inc.
jsquyres@cisco.com

Abstract. HPC systems are growing in both complexity and size, increasing the opportunity for system failures. Checkpoint and restart techniques are one of many fault tolerance techniques developed for such adverse runtime conditions. Because of the variety of available approaches for checkpoint and restart, HPC system libraries, such as MPI, seeking to incorporate these techniques would benefit greatly from a portable, extensible checkpoint and restart framework. This paper presents a specification for such a framework in Open MPI that allows for the integration of a variety of checkpoint/restart systems and protocols. The modular design of the framework allows researchers to contribute to specialized areas without requiring knowledge of the entirety of the code base.

1 Introduction

As High Performance Computing (HPC) systems grow in both complexity and size, they suffer from increased opportunity for system failures of various kinds. In fact, IBM warns users of the Blue Gene/L supercomputer that "faults are expected to be the norm rather than the exception" [11]. It is reasonable to expect that this advice will soon apply to other HPC systems as well. To run successfully on emerging large-scale platforms, HPC applications must become robust enough to account for their adverse operating conditions.

Modern HPC applications generally rely on message passing libraries such as the Message Passing Interface (MPI) for inter-process communication [10]. MPI, in turn, depends upon a parallel runtime system to manage the launching and coordination among processes in a parallel job. Fault tolerance should not (and, in reality, can not) solely be the responsibility of the application. System libraries such as MPI and its corresponding run-time environment can help account for and adapt to failures.

Checkpoint/restart systems are used to provide system layer support for checkpoint and restart fault tolerance techniques. These systems capture an image (or *snapshot*) of a running process and preserve it for later recovery. Checkpoint/restart coordination protocols generate a *global snapshot* of a parallel application by taking the union of the individual snapshots accounting for the global state of the parallel process [3,5]. Since the runtime system manages all of the processes in a parallel job, it is well positioned to assist in the generation of the global snapshots. There may be some state in a parallel application that is difficult or impossible for a checkpoint/restart system to preserve, such as shared memory regions, messages "in flight", or socket connections to remote machines. In many cases, this state is internal to an MPI implementation and could be preserved (or otherwise accounted for) by an MPI implementation that can interface with checkpoint/restart systems.

This paper presents a the requirements for the integration of checkpoint and restart fault tolerance techniques into Open MPI [7]. It provides a simple API for interacting with checkpoint/restart systems, and opportunity for incorporating a variety of parallel checkpoint/restart protocols to create global snapshots.

2 Related Work

A checkpoint/restart system is responsible for saving the current state of a sequential process for later restart (e.g., if that process is terminated by a system failure). Many checkpoint/restart system implementations are available, such as: libckpt [12], the checkpoint/restart system integrated into Condor [9], CRAK (Checkpoint/Restart As a Kernel module) [16], and BLCR (Berkley Lab's Checkpoint/Restart) [4]. These checkpoint/restart system implementations differ in many ways, including the method used to preserve the process state, how the state is stored, how much of the process state is preserved, APIs, and command line interfaces.

Checkpointing and restarting distributed or parallel applications may require additional coordination between individual processes to create a consistent checkpoint of the entire application. Coordinated and uncoordinated checkpoint/restart protocols are two such methods [5].

A few MPI libraries have attempted to integrate fault tolerance techniques. The techniques integrated range from user interactive process fault tolerance (FT-MPI [6]) to network failures recovery (LA-MPI [8]). 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. MPICH-V [2] uses an uncoordinated checkpoint/restart protocol to incorporates checkpoint/restart systems with message logging to account for process state. CoCheck [15] uses a coordinated checkpoint/restart protocol and the Condor checkpoint/restart system.

However, many of these MPI 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 to its code base [14]. But 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 checkpoint/restart systems provide a notification to `mpirun` in order to initiate the checkpoint of the parallel job.

3 Open MPI General Architecture

Open MPI consists of three abstraction layers that combine to provide a full featured MPI implementation, as illustrated by Fig. 1. Below the user application is the Open MPI (OMPI) layer that presents the application with the expected MPI specified 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 maximum portability. Below OPAL is the checkpoint/restart system available for the operating system running on the machine.

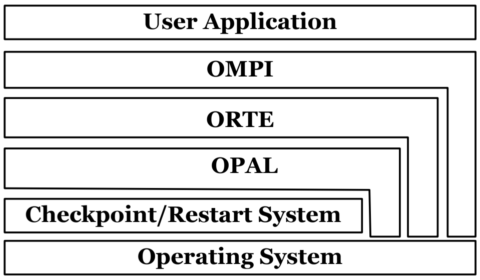

Fig. 1. The layered design of Open MPI with respect to the user application and checkpoint/restart system

Our checkpoint/restart work with Open MPI extends our previous work with LAM/MPI [13]. We retain `mpirun` as the central checkpoint/restart coordination point between the user and the parallel application but also allow other checkpoint mechanisms. The user is thus provided with a common reference no matter how the parallel application is deployed. The framework proposed in this paper is designed to allow any kind of application-level checkpointing scheme (e.g., both coordinated and uncoordinated types of protocols can be used). Also, Open MPI relaxes LAM/MPI's requirement that the checkpoint/restart system provide application-level notications when checkpoints occur. Instead, Open MPI provides and coordinates these notifications internally.

Open MPI uses the Modular Component Architecture (MCA) to define the OPAL Checkpoint and Restart Service (CRS) framework as a uniform API for checkpoint/restart systems. The OPAL CRS design allows for checkpoint/restart of MPI applications running on heterogeneous systems, even when multiple checkpoint systems are employed.

4 Open MPI Requirements

Open MPI requires the ability to support multiple checkpoint/restart services (e.g., BLCR, libckpt) and a variety of checkpoint/restart protocols (e.g., coordinated, uncoordinated).

4.1 Checkpoint/Restart System

The checkpoint/restart system is responsible for accurately preserving and restoring the state of a single process on a single machine. It may choose not to save static data areas, such as the program text, in order to reduce the size of and time taken to generate the checkpoint image(s).

Once the checkpoint/restart system has completed a checkpoint, it must provide Open MPI with a structure containing a reference to the checkpoint image (or images) generated, denoted by the term *snapshot reference* in this paper. The contents of the *snapshot reference* are determined by the checkpoint/restart system.

4.2 Checkpoint/Restart Protocol

Open MPI uses the snapshot references from all of the processes to create a *global snapshot* of the user application. The creation of the global snapshot is determined by the *checkpoint/restart protocol*. By interacting with the snapshot references instead of the checkpoint/restart system specific files, the checkpoint/restart protocol is abstracted from the underlying details of the checkpoint/restart system. This enables Open MPI to combine snapshot references from different checkpoint/restart systems into a single global snapshot, allowing checkpoints on heterogeneous systems. Further, by using the global snapshot, Open MPI could arrange for the migration of a single process or the storage of the global snapshot to a remote server, without requiring knowledge of the checkpoint/restart system used or how the checkpoint image(s) have been preserved.

Open MPI has some internal state that may not be accounted for by the back-end checkpoint system, such as shared memory regions (which should only be checkpointed by a single process, not all processes that share it) and network connections (that will be stale upon restart). Higher-level algorithms must coordinate between processes to capture a globally-consistent snapshot that either excludes this kind of data or invalidates it upon restart.

5 Handling Checkpoint Requests

An external checkpoint request is generated by a supporting tool sending the request to the `mpirun` command (see Section 7 for more details). Internal checkpoint requests are generated by `mpirun` distributing the request to the target parallel process. All of these processes handle the request by entering the OPAL `ENTRY_POINT` function, as illustrated by Fig. 2.

Checkpoint requests are handled by the OPAL `ENTRY_POINT` function in `mpirun`. Since different layers of the Open MPI hierarchy require the opportunity to prepare for and recover from a checkpoint, each layer can register an *intra-layer coordination callback* function. This function is called before and after a checkpoint is taken by the `ENTRY_POINT` function. By default the intra-layer coordination callback function is set to the OPAL coordination routine. It is then overridden by ORTE, and subsequently overridden by OMPI. Similar to POSIX signal handlers, the overriding layer assumes responsibility for calling the coor-

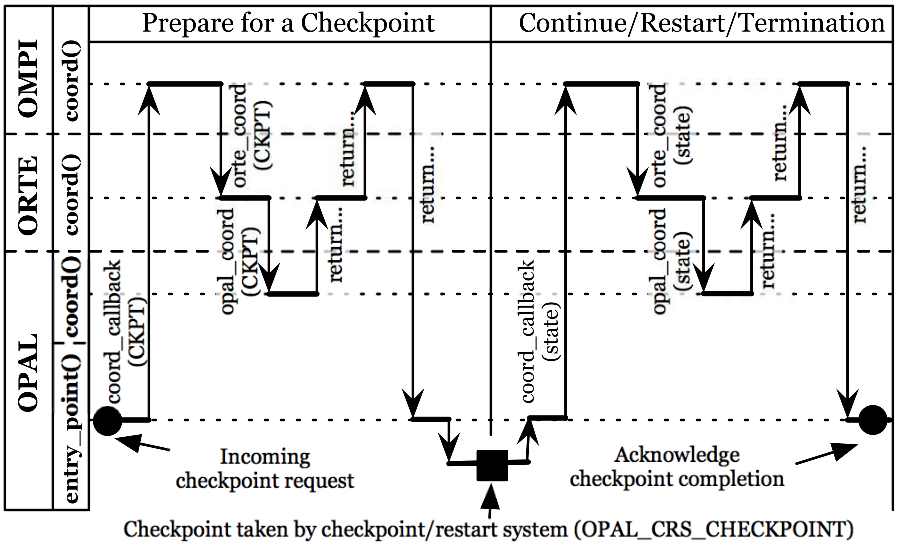

Fig. 2. Illustration of Checkpoint Request Handling in Open MPI

dination routine that it has overridden. Therefore the OMPI layer intra-layer coordination callback will call the ORTE layer intra-layer coordination callback before returning. Further the ORTE layer intra-layer coordination callback will call the OPAL layer intra-layer coordination callback before returning as illustrated by Fig. 2

When a checkpoint request arrives at a process, the OPAL `ENTRY_POINT` function first calls intra-layer coordination callback with the `CHECKPOINT` state indicating that the layers are to prepare for a checkpoint.

Once the coordination function has finished, the OPAL `ENTRY_POINT` initiates the checkpoint by using the `OPAL_CRS_CHECKPOINT` function. Through Open MPI's component system, `CHECKPOINT` invokes the selected back-end checkpoint/restart system to begin the process checkpoint. The return from the back-end checkpoint function will either be in the same process from which the checkpoint was initiated (known as the `CONTINUE` state) or will be in a new, restarted process (known as the `RESTART` state). Specifically, restarted processes do not start at `main()` – they simply "return" out of the back-end checkpoint function. If the checkpoint request indicated that the process should terminate after the checkpoint, then the notification routine changes the state to `TERMINATE`.

To allow the layers to recover from a checkpoint, the OPAL `ENTRY_POINT` function calls the intra-layer coordination callback again passing it the state returned by the `OPAL_CRS_CHECKPOINT` function. Once the OPAL `ENTRY_POINT` function is finished, the user application either resumes normal execution, or if the checkpoint request indicated that the application should terminate, exits the application.

In Open MPI, the checkpoint/restart protocol is integrated into the ORTE layer intra-layer coordination callback. This allows ORTE to coordinate as appropriate with the other processes in the parallel application to generate the global snapshot of it.

6 Checkpoint and Restart Service (CRS) Framework

The OPAL CRS MCA framework provides a simple API (shown in Fig. 3) for the Open MPI layers to interact with checkpoint/restart services. Every supported checkpoint/restart system creates a component of the CRS framework containing checkpoint/restart system specific commands as to conform to it. The OPAL CRS framework API extends the LAM/MPI API by removing the requirement that the checkpoint/restart system must provide an application notification upon a checkpoint. As such, Open MPI can support a wider variety of checkpoint/restart systems and more platforms. By adding the `CHECKPOINT` and `RESTART` functions to the API, this enables Open MPI to request a checkpoint internally as well as still retaining support for user requested checkpoints via command line tools (see Section 7 for more details).

```
int CHECKPOINT( pid_t              pid,
                snapshot_handle_t *snapshot,
                int               *state);
int RESTART(    snapshot_handle_t *snapshot,
                bool               spawn_child,
                pid_t             *child_pid);
int DISABLE_CHECKPOINT( void );
int ENABLE_CHECKPOINT(  void );
```

Fig. 3. OPAL CRS Framework API

The `CHECKPOINT` function initiates the checkpoint of a single process, identified by its PID, by calling the checkpoint/restart system's checkpoint routine(s). This function returns a `snapshot_handle_t` representing the *snapshot reference*. This function also returns the *state* of the system following the checkpoint that is used by the *inter-layer coordination callbacks*. The *state* is expected to be one either `CONTINUE` or `RESTART`.

The `RESTART` function initiates the restart of a single process from a *snapshot reference* by interacting with the checkpoint/restart system's restart functionality. The *spawn_child* argument indicates whether the checkpoint system should replace the current process image with the restarted process, or to spawn a child process and return the PID of the child.

Finally, the `DISABLE_CHECKPOINT` and `ENABLE_CHECKPOINT` functions can be used to surround critical sections of code where checkpoints should be disallowed (e.g., during MPI_INIT and MPI_FINALIZE).

7 Supporting Tools

Open MPI requires support for user or system service (e.g., a batch scheduler) directed checkpointing of MPI applications. Two command line tools are provided for this purpose: `ompi_checkpoint` and `ompi_restart`.

To send a checkpoint request to `mpirun`, the user specifies the PID of the application to the `ompi_checkpoint` command:

```
shell$ ompi_checkpoint [OPTIONS] mpirun_pid
```

When this command completes, the user is presented with a string name referencing the *global snapshot* that can be used to restart the parallel application.

To restart the parallel application (or a subset of processes from it), the user specifies the global snapshot name to the `ompi_restart` command, as seen below.

```
shell$ ompi_restart [OPTIONS] global_snapshot_reference_name
```

8 Summary and Future Work

Checkpoint and restart techniques are one of many fault tolerance techniques used by application developers. This paper presents an overview for integrating checkpoint and restart systems into Open MPI. These systems can than be used by upper-level protocols (such as in ORTE) to effect whole- or partial-job checkpointing and restarting, different protocols for creating (and maintaining) global snapshots, and whole- or partial job migration.

By logically separating the checkpoint of a single process from the checkpoint of an entire job, adding support for a particular checkpoint/restart system is both easy and orthogonal from complicated upper-layer protocols to effect parallel checkpoints. Perhaps more importantly, it also allows third-party researchers to continue studying checkpoint protocols independent of the back-end checkpointer that is used, allowing their work to be applicable to a wide variety of systems.

The framework described in this paper has been implemented in Open MPI and has integrated with BLCR and a "self" checkpointer (where user-level functions are called to write and read critical process state to effect the checkpoint). Support for more checkpointers will likely be added over time.

Future developments of this specification may include a protocol describing the movement of the checkpoint image(s) referenced by a snapshot reference to other machines, such as a checkpoint server. Other future developments may involve extensions to the API to enable explicit checkpoint image garbage collection requests, and explicit memory region inclusion and exclusion routines. Such API additions are meant to shrink the memory requirements for archiving or producing checkpoint images.

Acknowledgments

Special thanks to Brian Barrett for helping revise this paper. This work was supported by a grant from the Lilly Endowment and National Science Foundation grants NSF ANI-0330620, CDA-0116050, and EIA-0202048.

References

[1] Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, 2003.

[2] George Bosilca et al. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *SC'2002 Conference CD*, Baltimore, MD, 2002. IEEE/ACM SIGARCH. pap298,LRI.

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

[4] Jason Duell, Paul Hargrove, and Eric Roman. The design and implementation of Berkeley Lab's linux Checkpoint/Restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, 2003.

[5] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[6] Graham E. Fagg and Jack Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, 2000. Springer-Verlag.

[7] E. Garbriel et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, 2004.

[8] R. L. Graham et al. A network-failure-tolerant message-passing system for terascale clusters. *International Journal of Parallel Programming*, 31(4), August 2003.

[9] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report CS-TR-199701346, University of Wisconsin, Madison, 1997.

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

[11] Gary L. Mullen-Schultz. Blue Gene/L: Application Development. Technical Report SG24-7179-01, IBM, December 1 2005.

[12] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. Technical report, Knoxville, TN, USA, 1994.

[13] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. Checkpoint-restart support system services interface (SSI) modules for LAM/MPI. Technical Report TR578, Indiana University, Computer Science Department, 2003.

[14] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[15] Georg Stellner. CoCheck: Checkpointing and process migration for MPI. In *IPPS '96: Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531, Washington, DC, USA, 1996. IEEE Computer Society.

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