# COORDINATED CHECKPOINT/RESTART PROCESS FAULT TOLERANCE FOR MPI APPLICATIONS ON HPC SYSTEMS

Joshua Hursey

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University
July, 2010

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

Andrew Lumsdaine, Ph.D.

Randall Bramley, Ph.D.

Arun Chauhan, Ph.D.

Beth Plale, Ph.D.

Frank Mueller, Ph.D.

June 30, 2010

To Dad and Mom for their unwavering support over these many years.

# Acknowledgements

A special thank you to my advisor, Andrew Lumsdaine, for providing me with a unique working environment full of opportunities and challenges that have molded me into the researcher that I am today. My research perspective has been strongly influenced by his dedication to open, repeatable scientific exploration. His ability to coalesce and analyze a vast range of scientific domains to highlight new research avenues continues to amaze me.

I would like to express my gratitude to my research committee for their support during the development of the research contained in this dissertation. Whether in a formal classroom setting or informal conversation their expertise has been an invaluable asset. This dissertation was shaped by their insightful questions and suggestions.

I thank my colleagues in the Computer Science department and Open Systems Laboratory at Indiana University. A special thanks to Jeffrey M. Squyres, Brian Barrett, Torsten Hoefler, DongInn Kim, Timothy I. Mattox, Joseph A. Cottam, Andrew Friedley, Abhishek Kulkarni, and Timothy Prins for their collaborative help over the years. I would also like to thank Laura Hopkins for helping me polish this dissertation.

I would like to thank the Open MPI, Berkeley Lab Checkpoint/Restart (BLCR), and MPI Testing Tool (MTT) developer communities for their patience and support without which this dissertation would not have been possible. In particular, I would like to thank Ralph Castain, Paul H. Hargrove, Jeffrey M. Squyres, Richard Graham, and Ethan Mallove for the many hours they have spent with me discussing various design decisions. I have learned so much from each of you, for which I am truly grateful.

Many thanks to the staff of the Computer Science department, Computer Systems Group, and Pervasive Technology Institute. Lucy Battersby, Debbie Canada, Becky Curtis, Sherry Kay, Rebecca Lowe, Ann Oxby, and Jacqueline Whaley have been great assets

in helping me throughout my graduate career. Rob Henderson, Shing-Shong (Bruce) Shei, and T.J. Jones are some of the best system administrators I have ever had the pleasure of working with. Thank you for your patience and support as I experimented and stressed Odin and Sif beyond their comfort zones.

I owe a debt of gratitude to Charles Peck for sparking my interest in both teaching and research. His energy and passion for education, research, and life has left a lasting impression on me. Thank you for helping me find my path and showing me how to have fun with what I do.

Graduate school has been a taxing endeavor full of both joy and pain. To all of my friends at Indiana University and around the world, thank you for your support during this process. In particular, I would like to thank Craig Shue and Andrew Kalafut for helping me celebrate the good times and get through the rough times.

To my dear wife, Samantha Foley, you have been and continue to be a steadfast source of love, support, encouragement and kindness. Thank you for keeping me grounded and reminding me to not take things too seriously. I love you very much, and appreciate you every day.

Finally, I would like to thank my family not only for their unwavering love and support, but for instilling in me a work ethic and a set of values without which I could not have come this far. I am forever grateful of the hard work and sacrifice of my parents and grandparents which has provided me the opportunity to achieve something uniquely significant in our family's history.

This work was made possible by the generous support of the following grants and institutions: the Lilly Endowment; Lawrence Berkeley National Laboratory; Los Alamos National Laboratory; Oak Ridge National Laboratory; National Science Foundation grants NSF ANI-0330620, CDA-0116050, and EIA-0202048; and U.S. Department of Energy grant DE-FC02-06ER25750ˆA003.

# Abstract

Scientists use advanced computing techniques to assist in answering the complex questions at the forefront of discovery. The High Performance Computing (HPC) scientific applications created by these scientists are running longer and scaling to larger systems. These applications must be able to tolerate the inevitable failure of a subset of processes (process failures) that occur as a result of pushing the reliability boundaries of HPC systems. HPC system reliability is emerging as a problem in future exascale systems where the time to failure is measured in minutes or hours instead of days or months. Resilient applications (i.e., applications that can continue to run despite process failures) depend on resilient communication and runtime environments to sustain the application across process failures. Unfortunately, these environments are uncommon and not typically present on HPC systems. In order to preserve performance, scalability, and scientific accuracy, a resilient application may choose the invasiveness of the recovery solution, from completely transparent to completely application-directed. Therefore, resilient communication and runtime environments must provide customizable fault recovery mechanisms.

Resilient applications often use rollback recovery techniques for fault tolerance: particularly popular are checkpoint/restart (C/R) techniques. HPC applications commonly use the Message Passing Interface (MPI) standard for communication. This thesis identifies a complete set of capabilities that compose to form a coordinated C/R infrastructure for MPI applications running on HPC systems. These capabilities, when integrated into an MPI implementation, provide applications with transparent, yet optionally application configurable, fault tolerance. By adding these capabilities to Open MPI we demonstrate support for C/R process fault tolerance, automatic recovery, proactive process migration, and parallel debugging. We also discuss how this infrastructure is being used to support further research into fault tolerance.

# Contents

# List of Tables

# List of Figures

# List of Acronyms

# 1

# Introduction

Scientists use High Performance Computing (HPC) systems to solve complex scientific problems that, due to memory or compute performance limitations, either cannot be solved or are impractical to solve on more traditional computing systems. These HPC systems provide scientific applications with the parallel processing tools and compute resources necessary to distribute the application across many compute resources. These tools allow applications to solve larger problems, potentially, in shorter periods of time. For an application, HPC system reliability is typically determined by the number of components in the system being used by that application. As that number of components increases, the system becomes less reliable because the opportunity for component failure increases with the number of

components active, although not necessarily linearly. Unfortunately, many complex scientific applications often exceed the reliability of a given HPC system. Administrators of large HPC systems often measure system reliability, in terms of Mean Time to Failure (MTTF), in days or weeks [**236**]. However, administrators of future exascale HPC systems will likely measure system reliability in minutes or hours, further exposing the application to the risk of failure during normal computation [**41**]. If the application is not prepared for such inevitable failures, then it runs the risk of losing the entire computation, which may have taken hours or days to generate. The HPC system reliability problem is becoming so prevalent that some HPC system manufactures are advising application developers to prepare for inevitable failure by incorporating fault tolerance techniques, such as rollback recovery, into the application [**94, 244**].

Applications that are prepared to handle inevitable failures are called resilient applications. Rollback recovery techniques, often used by resilient applications, allow the application to take proactive action during normal computation, the results of which can be used to recover the computation after a failure. Checkpoint/Restart (C/R) is a particularly popular rollback recovery fault tolerance technique. C/R techniques periodically save application computational state information to a stable storage device during normal computation. Upon failure, the state information can be accessed from stable storage to recover the computation. Ideally, the C/R implementation would be transparent to the application, thus reducing the complexity of the application code. In order to provide a completely transparent C/R implementation on HPC systems, a set of capabilities must be organized in the underlying support software (i.e., Message Passing Interface (MPI) and runtime environments) to ensure correctness and consistency of the checkpoint operation. Previous C/R implementations have often struggled to remain relevant research platforms due to the lack of an extensible architecture. Such an architecture is only formed by first clearly identifying a complete set of required C/R capabilities. Transparent C/R solutions on HPC systems often use a fully coordinated technique to guarantee C/R consistency. Applications can usually improve the performance of the C/R operation by providing optional hints and

guidance to the support software regarding application state. However, a transparent, coordinated C/R solution will operate correctly without such guidance from the application.

Users of scientific applications and HPC system administrators, so called *end users*, often request the following three services from a C/R implementation:

- Reactive fault recovery
- Proactive process migration
- Parallel debugging assistance

Reactive fault recovery allows an application to automatically recover from a failure while running, or manually recover from job termination due to time limitations on a given HPC system. Proactive process migration allows an application to avoid predicted future failure by moving to a more reliable system during normal computation. C/R-enabled parallel debugging reduces the time spent debugging long-running applications by returning a developer to an intermediary point in the computation closer to the bug.

As can be seen in Chapter 2, previous research has investigated techniques for implementing the whole or part of each of these services. Unfortunately, no previous single implementation has presented an organizational infrastructure with clearly defined capabilities that can provide all three services. When we approached the problem of adding C/R support to the Open MPI project we wanted to design a solution that was maintainable for developers, extensible for researchers, and provided end users with the three services previously mentioned. From previous research and our own experimentation, we identified a complete set of C/R capabilities that, when organized appropriately, achieve those project goals.

In the context of this dissertation, a *capability* is a distinguishable abstraction in the C/R system design. In this dissertation we present an organization structure for the following identified C/R-related capabilities that compose to provide MPI applications with a transparent, but optionally configurable, coordinated C/R solution:

- Checkpoint/Restart Service (CRS): The interface to the single process C/R system provided by or for the system in order to capture an image of a running process for later recovery.
- Checkpoint/Restart Coordination Protocol (CRCP): The C/R coordination protocol implementation that marshals the network state to guarantee a consistently recoverable distributed state upon restart [**48**].
- Interlayer Notification Callback (INC): Notifying and coordinating subsystems of the MPI implementation around various checkpoint related activities (e.g., checkpoint, restart, migration).
- Stable Storage (SStore): A logical stable storage device abstraction encapsulating where and when local snapshots are stored in the distributed environment to form a global snapshot.
- File Management (FileM): The movement of snapshot related files and directories to and from storage devices possibly across file system and node visibility boundaries.
- Snapshot Coordination (SnapC): Checkpoint life-cycle management: distributing the checkpoint request to all participating processes, monitoring their progress, and synchronizing the final local snapshots to a logical stable storage device.
- Error Management and Recovery Policy (ErrMgr): Error reporting and fault recovery management operations including support for preventative actions such as process migration.

Identifying the necessary capabilities that form a C/R fault tolerance implementation will assist future implementations in creating more flexible designs and continue to support research innovation into C/R fault tolerance. Even though a C/R implementation may choose to combine one or more capabilities together, in Open MPI we demonstrated that a C/R implementation can support the three identified services without making such a compromise. Additionally, the seven capabilities presented in this dissertation can also be used to support research into alternative fault tolerance techniques beyond coordinated C/R such

as run-though stabilization, replication, and message logging. As HPC reliability declines, research into fault tolerance techniques must quickly adapt to the needs of the applications on these systems. A composable design with clearly identifiable capabilities allows C/R researchers to match their research efforts to the pace of the application's reliability requirements.

By integrating the seven capabilities into the Open MPI project we confirmed that they form a complete, functional set that is able to support real MPI scientific applications. There is often a performance cost to adding fault tolerance capabilities to an application. In this dissertation we investigated the performance implications of the C/R implementation in Open MPI on a variety of benchmarks and real applications including High-Performance Linpack (HPL), Parallel Ocean Program (POP) and Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS). The Open MPI implementation takes care to preserve performance across process recovery by uniquely allowing rediscovery of the interconnects between processes upon recovery. The time to save a checkpoint to stable storage is the predominate performance bottleneck in a transparent C/R solution. We present an analysis of a checkpoint staging technique that reduces checkpoint overhead by overlapping checkpoint establishment with normal computation. We also investigated the impact of checkpoint caching and compression on checkpoint overhead and latency. A novel, composable implementation of the ErrMgr capability allows applications to tailor recovery techniques at run time to best support their requirements.

Finally, we developed a set of Application Programming Interfaces (APIs) and command line tools for Open MPI that were designed to increase end user adoption and third party software integration. Applications are provided a set of optional API functions to guide the C/R related activities from within a process. These tools and APIs provided are designed to allow end users to interact with the C/R solution without requiring them to know all of the details about how it was deployed on a particular HPC system.

# 2

# Background and Related Work

*Reliable computing* techniques focus on providing a computing environment that can be trusted to work within expected parameters, usually up to a given time bound [**14**]. *Resilient computing* extends this bound by transparently detecting and recovering from a defined set of failures [**6, 14**]. Beyond this defined set of failures a best effort attempt is made to sustain the computing environment either through graceful degradation (a.k.a. partial fault-tolerance, or fail-soft operation) or by breaking transparency by involving the application in recovery. The reliability of a system is usually measured by the *mean-time-to-failure (MTTF)* which is defined as the expected time of normal operation between two consecutive faults [**1**].

*Fault intolerance* attempts to reduce the unreliability of the computing environment before the application begins execution [**14**]. This usually involves acquiring the most reliable hardware and software components at the time of construction, maintaining these components over the life of the computing environment, and developing techniques for initializing these components just before application launch in order to provide a reliable computing environment during the entirety of the application execution. Fault intolerance techniques support reliable computing by providing applications with bounds on the sustained reliability the computing environment can provide at the beginning of application execution with a high degree of probability.

*Fault tolerance* uses protective techniques to provide a resilient computing environment in the presence of an expected set of component failures [**14**]. Protective techniques to provide fault tolerance can be classified into two broad categories: *Algorithm-Based Fault Tolerance (ABFT)* and *redundancy*. ABFT focus on choosing algorithms that can withstand the failure of one or more computational tasks and continue computation by working around the loss of the data operated on by the lost computational task(s). ABFT approaches may employ some redundancy techniques, but are not strictly required to do so.

Redundancy is a commonly employed technique for providing fault tolerance. Hardware redundancy relies on multiple, similar physical components that can be activated in order to recover transparently from the failure of the peer component [**15**]. Processor hardware redundancy is one example where a multi-processor system allows for the loss or replacement of a physical processor while the system is running by transferring all tasks in execution from the lost processor to others on the system without any loss in transparency to the application. Redundant Array of Independent Disks (RAID) [**202**] based techniques have also been designed to provide transparent failure recovery of hard disks by managing redundant physical disks.

Software redundancy relies on the replication of a computational task that can be used to continue operation when one or more of the replica tasks have been lost due to component failure in the computing environment [**6, 157**]. The maintenance of the replicas typically involves distributed election and consensus algorithms that grow in complexity in

response to not only the size of the computing environment but also in the number and types of failures that need to be handled.

Time-based redundancy or rollback recovery techniques rely on the re-execution of all or part of the application in order to recover from a component failure in the computing environment. In checkpoint and restart rollback recovery, *snapshots* (or checkpoints) of the application state are saved to a *stable storage* device during normal execution. A *stable storage* device is a logical device that survives the maximum number of failures in the system. These checkpoints are then used to restore the application to a previous execution state after a failure [**141**]. The application must re-execute the amount of lost work between the state restored from the checkpoint up to the point of the failure. In a distributed computing environment where individual processes interact with one another through various events (typically Inter-Process Communication (IPC)) care must be taken to maintain transparency of recovery of one or more tasks in the context of a larger dependent network of failure-free processes.

Event logging rollback recovery techniques focus on such interactions by writing to stable storage the contents and/or ordering of all external events that influence the execution of the process in order to deterministically replay these events upon re-execution of the failed task. Message logging is a commonly used sub-domain of event logging that focuses just on the logging of messages between processes in the system. When used alone event logging techniques require the re-execution of the entire task. Since this can cause excessive time delays during recovery, event logging is usually combined with checkpoint/restart techniques to resume re-execution from an intermediary computational state instead of the beginning of execution.

### 1. Distributed Fault Detection

In a single system, an *error* is generated when a physical defect, called a *fault*, is detected. A system *failure* is when the system cannot deliver its intended function because of one or more errors [**1**]. A fault-tolerant system will continue to operate normally in the presence of errors. Individual faults are generally classified into one of three categories:

permanent, transient, and intermittent [**1**]. A *permanent fault* is a fault that continues to exists until it is repaired. A *transient fault* is a fault that occurs and disappears at an unknown frequency. An *intermittent fault* is a fault that occurs and disappears at a known frequency.

A distributed system is composed of many systems each with one or more processes working together towards a shared goal. The failure of one process in a system can lead to the failure of the entire distributed system. A process failure is often classified in one of three categories: fail-stop, omission, and Byzantine [**80**]. A *fail-stop* failure is when a process is permanently stopped, often due to a crash. An *omission* failure is when a process fails to send or receive messages correctly. A *Byzantine* failure is when a process continues operating but propagates erroneous messages or data [**159**]. Byzantine failures are often caused by undetected faults or mishandled errors in the system. *Soft errors* are transient errors often caused by radiation [**180**]. Soft errors are often seen in Random Access Memory (RAM) and protected using Error Correcting Codes (ECC) [**294**], but has not been well addressed in other parts of the system including the CPU [**180**]. Since soft errors can be difficult to detect they often manifest themselves as Byzantine process faults. Since Byzantine failures are difficult and costly to detect in a distributed system [**80**], we focus our attention in this dissertation on permanent, fail-stop process failure in this dissertation.

The ability to detect a fail-stop failure in a distributed system is central to any fault tolerant solution. It has been shown that it is impossible to accurately detect even a single failure in an asynchronous system [**98**] because a failed process is indistinguishable from a process that is running very slowly. Branching from the impossibility result in [**98**], researchers have explored fault detection mechanisms that allow for unreliable failure detectors [**47**] and partially synchronous systems [**72, 80**]. The aspects critical to any fault detection algorithm are those of completeness and accuracy as outlined by [**47**]. *Completeness* requires a failure detector in the system to eventually suspect every process that actually crashes. *Accuracy* restricts the mistakes that the failure detector can make during the life of the system.

The fundamental programmatic building block of every fault detection mechanism is that of a *push* or *pull* model. In a *push* model, a message is sent from a process to its monitor in a regular, predefined schedule. In a *pull* model, a message is sent from the monitor to a process, and the process must then respond with a message in a defined amount of time. The pull model contains more overhead in terms of the number of messages that need to be generated per process, but it is able to detect additional failure modes, such as network partition, that push models cannot.

Building on these fundamental concepts, research into scalable fault detection mechanisms seeks to reduce the number of processes being monitored by any single process while still providing high confidence in the fault detection mechanism in terms of completeness, accuracy, and performance. Gossip-style failure detectors distribute fault information to a random (or pseudo-random) limited set of peers that then propagate this information to another random set of peers [**220, 271**]. Gossip-style failure detectors are closely related to randomization fault detection techniques [**121**] since each build upon an epidemic computational model [**69**].

Hierarchical failure detectors have also been explored to increase performance in a large-scale system by employing a failure detection algorithm through each level of the hierarchy [**20**]. This research also explores using different fault detection mechanisms at different levels in the hierarchy in order to balance performance and completeness requirements.

## 2. Fault Prediction

The ability to predict impending failure, and take preventative action, is central to the topic of proactive fault tolerance which usually involves migrating processes away from the predicted failure [**90, 267, 277**]. Prediction techniques usually base their models on historical data and real-time events. Historical data is usually derived from mining event logs [**124, 199, 206, 236**]. Real-time events are usually derived from hardware sensors, which are becoming more prevalent in computing hardware [**100**].

The prediction algorithms that use these data can take a variety of forms [**50, 232**]. They must be able to determine both when and where a failure is going to occur with a sufficiently high degree of accuracy [**290**]. Typically a fault prediction service depends on a separate process in the system to determine the appropriate action to avoid the failure, for example initiate process migration [**168**]. This process must account for the additional stress on the system caused by this preventative action, which might affect the reliability of the system.

It has been suggested that there is a strong need for system-level monitoring and prediction services in large HPC systems in order to support fault tolerance activities [**223**]. It has also been noticed that a failure predictor building upon such information can have a significant impact upon system reliability even at low prediction accuracies [**197**].

## 3. Fault Recovery: Algorithm Based Fault Tolerance

Algorithm-Based Fault Tolerance (ABFT) techniques require specialized algorithms that are able to adapt to and recover from process loss due to hardware failure [**131**]. ABFT techniques typically require data encoding, algorithm redesign, and diskless checkpointing [**207**]. These techniques require a resilient message passing environment (e.g., FT-MPI [**93**]) that can continue running when a process is lost, and possibly allow the recovery of the lost process. Matrix operations have been the focus of many applications of ABFT [**52, 53, 54, 75, 131, 149, 160**], though recently it has been applied to heat transfer problems [**171**]. Manager/worker programming techniques could be classified as ABFT since the loss of a worker process can be recovered by the manager if it maintains the work unit assigned to the worker [**164**].

The methods for generating the data encoding or checksum must consider the algorithm that they are encoding, the degree of fault tolerance required (e.g., the number of acceptable concurrent failures it can handle), and performance [**53, 54**]. Some algorithms forgo data encoding by taking advantage of the ability to recalculate the values from the current solution due to special properties of the algorithm [**75**]. The method of storing the data

encoding on peer processors (also known as *diskless checkpointing*) affects the performance and degree of fault tolerance of the implementation [**55, 149**].

There is a slight difference between ABFT and natural fault tolerance described in [**88, 107**]. Both require algorithm changes to prepare for process failure. ABFT use a combination of data encoding and diskless checkpointing to preserve state that could be lost. Natural fault tolerance techniques focus on algorithms that can withstand the loss of a process and still get an approximately correct answer, usually without the use of data encoding or checkpointing. So natural fault tolerance can be viewed as a more general form of ABFT.

## 4. Fault Recovery: Checkpoint/Restart

Checkpoint and restart rollback recovery is a technique used to reduce the amount of computation lost to process failure by restoring the computation to a previously-established point in the computation. Applications establish checkpoints during failure-free operation by writing them to a stable storage device. A *stable storage* device is a logical device that survives the maximum number of failures in the system. Usually this is represented as a centralized file server for recovery of all processes, though peer-based techniques can be used for partial recovery.

Since fault tolerance techniques distract an application from normal execution they come at an additional cost in terms of application performance. When discussing the cost of checkpoint/restart fault tolerance techniques, we must consider the effect on both the application and the system. The additional execution time required by the application as a result of introducing checkpointing techniques is called the *Checkpoint Overhead* [**206, 265**]. The *Checkpoint Latency* is the time required to create and establish a checkpoint to stable storage [**206, 265**]. If the application is suspended until the checkpoint is established then the checkpoint overhead is equal to the latency.

C/R and stable storage techniques that overlap checkpoint establishment with application execution can improve application performance. To describe the overhead involved with these techniques it is important to distinguish between checkpoint overhead and latency. *Forked* checkpointing is one example of such a technique, based on copy-on-write

semantics of modern operating systems, which suspends a child process for checkpoint while the parent continues execution [**265**]. Reductions in checkpoint overhead result in the largest performance gain to the application by overlapping the establishment of the checkpoint with program execution. However, the interference of these techniques with the application must be accounted for. Interference as a result of potentially sharing a common processor, paging overheads, I/O storage requirements, and impact on shared resources must all be considered when choosing checkpointing techniques and intervals.

**4.1. Checkpoint Interval.** If the checkpointing overhead were negligible, then the optimal checkpointing strategy would be to checkpoint after every instruction [**217**]. However, depending on the size of the application state and the checkpointing techniques used, the overhead can become significant. The frequency of checkpointing must be adjusted and modeled appropriately in relation to the checkpoint overhead. The *Checkpoint Interval* is the time between the establishment of two consecutive checkpoints [**265**]. Over the years many models have been proposed for choosing the optimal checkpointing interval [**63, 64, 77, 108, 109, 141, 265**]. It has been noted that the checkpoint interval is usually independent of the checkpoint latency, and strongly dependent upon the checkpoint overhead [**206, 265**]. Most checkpointing interval models assume a Poisson fault distribution and a static checkpointing interval. In recent years, these two assumptions have been reconsidered [**236**].

Through the study of system logs researchers have found that a Poisson fault distribution model does not accurately represent the failure profiles of the systems considered [**199, 206, 236**]. Instead studies suggest that a Weibull or gamma distribution serve as better models for system failures [**124**]. As a counterpoint, it has been shown through simulation that two of the checkpoint interval models designed with Poisson fault distributions in mind, namely [**265**] and [**285**], also perform well with non-Poisson fault distributions [**206**]. It has been also highlighted that the failures in these systems can be clustered both in time and in space suggesting that a single failure has an immediate impact on the components (e.g., machines, power supplies) physically located near the effected component [**100, 124**].

Since checkpointing incurs some overhead the optimal placement of a checkpoint is just before a failure. Unfortunately without a perfect failure predictor, checkpoints must be taken that will, with high likelihood, never be used in recovery, so called *useless checkpoints*, or else run the risk of total loss of the computation. This realization and the refinement of failure models and predication frameworks has brought into question the static checkpointing interval. Since these models are not completely accurate, most modern checkpoint interval models combine a periodic checkpoint with a fault aware, probabilistic checkpointing interval [**218, 256**]. The periodic checkpoint interval provides some protection against inaccurate failure models, while the fault aware checkpoint interval attempts to checkpoint just before a failure, preserving the maximum amount of program state for later recovery. Other studies suggest that a dynamic checkpoint interval based on fault prediction or application size can improve the checkpointing overhead by reducing load on shared resources, namely stable storage [**13, 198**]. Some systems track the dynamic memory allocation/deallocation and adjust the checkpointing interval in response to the application's memory footprint, thus reducing the size of the checkpoint and the resultant I/O requirements [**166, 295**]. These techniques show lower checkpoint overhead than fixed checkpoint interval algorithms. Even without dynamic checkpoint interval selection, re-evaluating the fixed checkpoint interval after recovery, based on fault history, has shown improvement [**231**].

Other important metrics to consider when building a checkpoint/restart system include recovery time, disk space consumed, fault coverage, impact on shared resources, and I/O requirements [**205, 206**]. When studying large-scale HPC systems the need for high sustainable I/O bandwidth becomes a critical concern both for *Productive I/O* (e.g., for scientific visualization) and *Defensive I/O* (e.g., for checkpoint/restart) [**252**].

**4.2. Checkpoint/Restart Services.** A Checkpoint/Restart Service (CRS) captures the state of a single process in execution. Used in combination with a parallel runtime and communication environment, a CRS can be used to take a checkpoint of a multiprocess application. CRSs are classified by their degree of transparency, portability, and location

of implementation. Though some literature will refer to application-level CRS as user-level CRS, for our discussion we will distinguish them by their transparency with regard to the application program, further classified below.

4.2.1. *Application-Level.* Application-level CRSs interact directly with the application to capture program state. This requires changes to the application code to identify regions of memory and variables that need to be preserved in a checkpoint in order to correctly restart the application. In Section 10, we will discuss how a pre-compiler can be used to automate the augmentation of the program to support application-level CRS.

Application-level CRSs are often able to create the smallest possible checkpoint since they can take advantage of knowledge about the program state [**239**]. This is at the cost of loss of transparency since application modifications are required. The consistency of the checkpoints in the distributed environment is completely left to the application to determine through normal messaging.

One of the major complications of such techniques is the non-standard nature of the interfaces to various CRSs. Some CRSs allow applications to use MPI datatypes for packing and unpacking data to and from a checkpoint service [**238, 255**]. Some also allow the user to use parallel I/O functionality, as defined in the MPI standard, though they have to define their own synchronization points, which can be difficult [**61**]. Most use callback functions during the pre-checkpoint and restart/continue phases to help the application encapsulate the checkpoint specific functionality [**3, 62, 87**]. Some restrict the structure and types of data that can be saved in the checkpointing functions [**261, 262**]. Some require special member functions in an object oriented programming paradigm [**17, 291**].

4.2.2. *User-Level.* A user-level CRS is implemented in user-space and typically provides transparency by virtualizing all system calls into the kernel. Within this virtualized environment the CRS is able to capture the state of the entire process without being tied to the kernel, as with system-level CRSs, but at the cost of a constant virtualization overhead during normal operations. In contrast with application-level CRSs, user-level CRSs produce larger checkpoints since they are not able to take advantage of optimizations based upon

application specific semantics. User-level CRSs are seen as more portable than system-level CRS since they are not tied to a specific kernel revision set, but often they generate checkpoints that are tied to a particular machine or kernel type since data segments can be difficult to manage in a portable manner.

Most user-level CRSs are loaded dynamically in the application by using dynamic libraries that can wrap the `main` function and system calls [**31, 246**]. The `main` function is often wrapped as a mechanism for initializing the CRS and restarting a process from a previous checkpoint. Many user-level CRSs struggle with interprocess communication, process hierarchies, shared memory or libraries, signals and timers since they can be difficult to properly virtualize [**132, 169**]. For interprocess communication, most solutions rely on callbacks into higher level message passing environments to support communication channels [**43, 61, 210, 254**]. Other solutions use specialized virtualization of the sockets interface to handle interprocess communication [**11, 12, 154, 230**].

Some implementations blur the line between user and application-level CRS by requiring the application to insert checkpointing function calls into their application in order to use an otherwise transparent user-level CRS service since they are unable to activate the service external to the process [**70, 71**].

When dealing with a multiprocess environment consistency between the checkpoints is important. It can be difficult, if not impossible, for an application to choose proper synchronization points to guarantee consistent checkpoints especially when the checkpoint consists of the entire process image. Highly structured Single-Process Multiple-Data (SPMD) applications with clearly defined synchronization points may be able to achieve this with a CRS that requires such synchronization [**114, 181**]

4.2.3. *System-Level.* A system-level CRS is implemented either inside the kernel or in a kernel module. It is able to directly access the kernel structures representing a process and associated threads, so there is little or no need to virtualize the system call interface. Without the virtualization overhead system-level CRSs do not suffer from the continual overhead of virtualization-based techniques during normal operation, at the cost of generating checkpoints that are often tied to specific kernel revisions. The system-level CRS must

constantly track the kernel development since changes to internal data structures can break functionality.

Kernel modules are a preferred method for implementing a system-level CRS [**78, 79, 99, 101, 113, 293**] since it requires less maintenance and increases user adoption than requiring changes to the kernel source code [**268**]. For interprocess communication, system-level CRSs use callbacks into the higher level communication libraries to handle the communication channel state [**37, 140, 234, 235**]. Operating system virtualization techniques used by Xen [**184**] and other environments [**155, 273, 275**] are also classified as system-level CRSs since they provide checkpoint/restart functionality in the operating system.

One of the challenges when dealing with a CRS is the interface which changes for each project. Recently an attempt has been made to unify the interfaces to support the non-uniform deployment of CRSs in grid systems [**177**].

**4.3. Checkpoint/Restart Optimization Techniques.** There are many optimizations that checkpoint/restart systems can take advantage of in order to reduce the checkpoint overhead and/or checkpoint latency. In this section we will explore a set of commonly used optimizations.

4.3.1. *Forked Checkpointing.* Forked checkpointing relies on the copy-on-write semantics of process creation in modern operating systems in order to reduce checkpoint overhead [**57, 153, 208, 238, 265**]. At the point of the checkpoint, the parent process `fork`s a child process to perform the checkpoint operation while the parent program continues execution. Since the child is only reading pages which are then written to stable storage as the checkpoint there is no need for a complete copy of the parents memory image, only those pages of memory that have changed since the `fork` operation. By using copy-on-write semantics the child process is able to be created quickly and run concurrently with the parent process, thus reducing the impact on checkpoint overhead.

The performance gains diminish if the parent process writes a large amount of data while the child process is writing the checkpoint due to paging overhead. Additionally, in memory constrained systems, where the application requires most or all of main memory

the child process may be forced to page to disk causing trashing which drastically diminishes the performance of both the checkpointing operation and the parent process execution. This technique works well for programs that do not use `fork` sensitive libraries or resources. Unfortunately, high performance interconnect drivers are typically sensitive to `fork` calls due to operating system bypass techniques used for optimized memory transfers making them ill suited for these techniques.

Similar to forked checkpointing is a *mirror copy* (a.k.a. continuous checkpointing) technique which seeks to reduce the overhead of copy-on-write paging by keeping a duplicate copy of the process image in memory at all times [**59, 127**]. This duplicate image is then occasionally written out to stable storage concurrently by the operating system [**59, 127**]. For real-time systems, the mirror copy technique provides a more consistent checkpoint latency than copy-on-write making it more appealing to system with hard deadlines on the completion of application execution [**59**].

Instead of maintaining this duplicate image throughout the life of the process, a *pre-copy* technique has been suggested that creates a full duplicate copy of the parent at the point of the checkpoint [**83, 257**]. It was shown by [**83**] that forked checkpointing performed better in practice than the *pre-copy* technique, indicating that the overhead of maintaining the copy-on-write properties of main memory is not a significant enough factor to warrant such an optimization.

4.3.2. *Checkpoint Compression.* One way to reduce the checkpoint latency is to reduce the amount of checkpoint data that is pushed to stable storage, thus reducing the I/O required to store the checkpoint [**166, 208, 236, 238**]. This technique may also reduce the amount of space required on stable storage for checkpoint files. Most of these techniques employ in-line compression while writing the checkpoint [**208**], but few have looked at compression as part of a larger staging process at the stable storage level.

The amount to which a checkpoint can be compressed is application state dependent. Most studies have shown that checkpoints, especially application generated checkpoints, can be significantly compressed. Though these studies typically focus on the size reductions, it is equally important to consider the performance of the checkpointing algorithm

as it impacts checkpoint overhead. [**208**] notes that while compression may reduce the size of the checkpoint, "... it only improves the overhead of checkpointing if the speed of compression is faster than the speed of disk writes, and if the checkpoint is significantly compressed." Informally, researchers have found that even when checkpoint overhead is increased by using compression, the reduction in stress on the stable storage system often improves system reliability allowing the application to checkpoint less frequently.

4.3.3. *Memory Exclusion.* Another method of reducing checkpoint latency is to exclude temporary or unused buffers from the checkpoint [**209**]. This can reduce the size of the checkpoint and therefore the amount of I/O required to transfer it to stable storage. Most of the techniques focus on augmenting applications to mark regions of memory that should be excluded from the checkpoint [**3, 51, 208**]. Alternatively the checkpointing library may be able to transparently find memory to exclude by augmenting the memory allocator [**51**] and/or using pre-compiler analysis [**175**].

This technique relies on the ability of the application writer to highlight both critical and temporary regions of memory. Usually this is employed by support libraries as a way to reduce their additive impact on the amount of additional state that needs to be saved in the checkpoint as a result of their use by the application [**3**].

4.3.4. *Incremental Checkpointing.* Incremental checkpointing techniques focus on reducing checkpoint latency by checkpointing only the changes made by the application from the last checkpoint [**208**]. During recovery the last N checkpoint files are required in order to recreate the process. In order to reduce the number of checkpoint files necessary for recovery, a full checkpoint is taken less often than the incremental checkpoints and used as a base for the next set of incremental checkpoints to be combined against. Similar to forked based checkpointing these techniques commonly rely on the paging system of modern operating systems, particularly the modified or dirty bit in modern paging hardware [**51, 83, 87, 95, 113, 127, 208, 215, 243, 273**]. When a checkpoint is taken the modified (a.k.a. dirty) bits are cleared, and as the program executes it flips the bits on the pages that have been modified. During the next pass of the CRS, only those pages that have

been modified are included in the incremental checkpoint saved to stable storage. Upon recovery, the incremental checkpoints are combined with the last full checkpoint to recreate the process state.

The smallest incremental checkpoint is based on the smallest segment of memory, the word. Since the overhead involved in tracking changes to individual words in memory is prohibitively expensive most techniques use pages which contain a set of words, and paging hardware to support incremental checkpointing at the page level. However even a single word change may cause the entire page to be included in the incremental checkpoint adding expense to the checkpoint operation. *Probabilistic checkpointing* allows for incremental checkpointing based on blocks of memory which are larger than a word, but smaller than a page in order to reduce this overhead [**3, 186**]. These techniques use a memory block encoding algorithm to determine if a block of memory has changed, since they cannot rely on paging hardware to highlight differences. The encoding technique explored often suffers from aliasing, in which differences may be masked because the encoding algorithm encodes the old and modified blocks to the same value. Though the authors of technique present analysis that shows this to be a sufficiently rare event [**186**], later research found that aliasing occurs much more frequently in practice making probabilistic techniques dangerous to employ [**81**].

**4.4. Checkpoint Coordination Protocols.** The state of a distributed system is composed of the state of the process and all connected channels [**48**]. In Section 4.2, we presented various techniques for capturing the state of a single process. In this section, we will discuss protocols to account for the channel state in a distributed checkpoint operation. [**192**] formally defines three communication channel states that can exist in a *global cut* formed from local checkpoints of two or more processes in a distributed system. These states are strongly based on the theoretical foundations of [**48**] and [**158**]. See Figure 2.1 for an illustration of these three communication channel states. A *consistent state* is a state in which there does not exist any message sent after the global cut, but received before it. A *transitless state* is a state in which there does not exist any message sent before the global

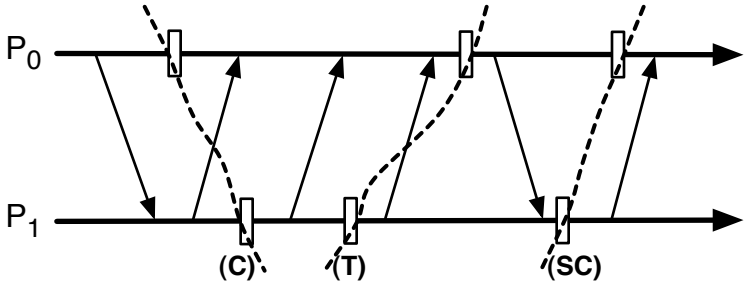



FIGURE 2.1. Illustration of the three communication channel states. White blocks indicate checkpoints of the individual processes. Dashed lines indicate global cuts of the system. (C) is a *consistent state*. (T) is a *transitless state*. (SC) is a *strongly consistent state*.

cut, but received after it. A *strongly consistent state* is a state that is both consistent and transitless, thus empty or quiescent at the point of the global cut.

RENEW [**193**] and Egida [**221**] each presented a set of minimal functional components that can be used to create a checkpoint/restart or message logging protocol. Our approach described in Chapter 3 was to create a virtualization layer and allow the protocol to be implemented as a single functional component not constrained by a predetermined the set of low-level fault tolerance APIs.

**4.5. Coordinated Protocols.** Coordinated checkpoint/restart coordination protocols require that all processes take a checkpoint at logically the same time. These protocols operate before a checkpoint is taken to ensure that the state of all connected channels is either consistent or strongly consistent between all processes. Coordinated protocols are relatively simple in nature, but are often critiqued for their synchronization overhead since all processes must participate in every checkpoint operation. Though many protocols exist for checkpoint/restart coordination, fully coordinated protocols are usually chosen due to relative simplicity in implementation (easier to prove that it is working correctly) and in recognition that the synchronization overhead is negligible in comparison with the storage overhead that dominates the checkpointing time [**82, 83, 84, 240**].

The *distributed snapshots* (a.k.a. Chandy/Lamport) algorithm allows a process to checkpoint once it has received a special marker token from every process in the system indicating

the point in logical time on the channel where communication between the two processes is consistent [**48**]. This method requires FIFO communication channels between all processes to form a consistent state. Since the system is assumed to be able to control the delivery of the messages, it is assumed that once the marker has been received all other messages on that channel can be delayed by the system until the checkpoint is established. Various systems have implemented the distributed snapshots algorithm usually with the assistance of a daemon to control message traffic to the process, thus delaying messages after the marker. Sometimes this is called a *non-blocking* coordinated checkpoint/restart protocol since the channel is not blocked once the marker is received [**30**]. MPVM [**43**], ickpt [**210**], MIST [**42**], ABARIS [**144**] and MPICH-Score [**105**] are examples of systems that use this technique.

The *ready message* algorithm refines the distributed snapshots algorithms by forcing the channels to be shutdown once the markers have been received to make a strongly consistent channel state [**248**]. The name for this protocol was established by the CoCheck implementation which supports both Parallel Virtual Machine (PVM) and MPI applications [**248**]. This protocol is useful in situations where the application cannot control the delivery of messages to the application, so it may accidentally include a message sent on the channel after the marker. An outside daemon is used to "break the silence" after a checkpoint operation has finished. Sometimes this is called a *blocking* coordinated checkpoint/restart protocol since the channel is blocked once the marker is received. Dynamite followed CoCheck by implementing this protocol for PVM with process migration capabilities [**139**]. The MPICH-GF project added a coordinated checkpoint/restart protocol to MPICH-G2 for grid based systems using Ethernet communication [**283**]. The FTMPI project implemented a synchronized checkpointing protocol in MPICH [**237**]. They used replicated checkpoint protocol control systems, and a checkpoint sever to support checkpointing activities. The $M^3$ project implemented a coordinated checkpointing protocol in MPICH-GM supporting Myrinet interconnects [**147**]. The MVAPICH project also implemented a coordinated checkpointing protocol but supported InfiniBand hardware [**103, 104**].

In shared memory systems a memory barrier is sometimes sufficient to synchronize the checkpoint if the entire memory space is written to disk as demonstrated by [**2**].

The MPICH-V [**29**] project compared pessimistic, sender-based message logging with a distributed snapshots coordinated checkpoint/restart algorithm (the MPICH-V2 and MPICH-Vcl implementations respectively). They concluded that for low fault frequencies the coordinated checkpoint algorithm performed better due to the reduction in runtime overhead [**30**]. A year later they expanded their testing to include causal message logging protocol (MPICH-Vcausal implementation) and made the same conclusion with regard to coordinated checkpointing [**163**]. The same project compared a blocking coordinated checkpoint/restart implementation (MPICH-Pcl) against their non-blocking coordinated checkpoint/restart implementation (MPICH-Vcl), and found that the blocking version provided the best performance overall, possibly due to the limited message logging overhead required by the non-blocking implementation [**38, 57**].

The $C^3$ project [**32**] uses a non-blocking coordinated protocol that adapts to different consistency states in the network channels by piggybacking data on all messages [**33, 34**]. Since they implement their protocol above the MPI layer they do not use the provided collective operations since a checkpoint may occur across a collective operation so they re-implement them using point-to-point operations [**35**]. This project also requires non-standard behavior from the MPI implementation by requiring the ability to call `MPI_INIT` and `MPI_FINALIZE` multiple times in order to generate a consistent checkpoint state for the application. The $PC^3$ project extends the $C^3$ project by adding support for heterogeneous checkpointing through application-level checkpointing and paying close attention to type and pointer portability [**96**].

*Sync-and-stop* style algorithms exchange message counts between all processes and drain the network to create a strongly consistent channel state between all processes across the checkpoint operation [**210**]. An outside daemon is uses to "break the silence" after a checkpoint operation as in the ready message protocol.

The LAM/MPI [**39**] project incorporated transparent checkpoint/restart functionality with support for Ethernet [**234**] and later Myrinet GM using the Berkeley Lab Checkpoint/-Restart (BLCR) library. They used an all-to-all bookmark exchange coordination algorithm implemented as part of the TCP/IP and Myrinet GM interconnect drivers. LAM/MPI was unique in its support for a modular interface to the checkpoint/restart services on a given machine [**233, 234**]. It supported BLCR and a user-level callback system called *SELF*.

The Open MPI project implemented a coordinated checkpointing protocol similar to the LAM/MPI implementation [**136**], but supports a wider variety of interconnects [**135**]. An automatic recovery technique called *job pause* (a.k.a. automatic, in-place recovery) was added to LAM/MPI that, upon detection of process failure, rolls all processes back to the last checkpoint and restarts the failed process on a spare machine [**276**]. Chapter 4 describes how this techniques was ported to Open MPI [**134**]. A similar sync-and-stop model was used to migrate PGAS processes using InfiniBand networks in a Xen virtual machine [**235**].

Transparent checkpoint/restart in MPICH-GM using the Myrinet GM driver has also been demonstrated [**147**]. They use a 2-phase coordination procedure based, in-part, on CoCheck's Ready Message protocol [**248**]. They implement their coordination algorithm as part of the GM driver relying on the First-In-First-Out (FIFO) message ordering provided therein. The GM Driver supports shared memory communication for peers on the same node. Even though their coordination algorithm is influenced by the Remote Data Memory Access (RDMA) semantics of the GM driver, they do support reconfiguration between shared memory and Myrinet GM communication, though they require Myrinet to be available upon restart to all processes in the application. Chapter 4 describes how by lifting the coordination protocol out of the device driver additional drivers and features can be realized [**134**].

The MPICH-V project focuses its effort towards message logging techniques with the MPICH-V1, MPICH-V2 and MPICH-Vcausal implementations [**29**]. They also have a MPICH-Vcl implementation that supports coordinated checkpoint/restart using a Chandy/Lamport coordination algorithm [**48**]. In their papers, they are able to demonstrate their implementations running on clusters with 100 Mbit/s Ethernet, Myrinet, and SCI networks [**57, 163**].

However, to run on the Myrinet and SCI networks, they take advantage of the Ethernet emulation provided by these interconnects and do not interact directly with the hardware drivers. Though this saves in the overall complexity of the solution, it comes at a significant performance loss.

MVAPICH2 demonstrated transparent MPI checkpoint/restart functionality over InfiniBand [**138**] interconnects [**104**]. This work highlights the complexity of dealing with the OS bypass technique used by InfiniBand. They show that the only way to properly handle such an interconnect driver is to completely shutdown all network connections before a checkpoint and re-establish them directly after a checkpoint. They implement a Chandy/Lamport [**48**] style coordination algorithm at the InfiniBand driver level operating on network-level messages instead of MPI-level messages. This coordination algorithm relies on FIFO message ordering provided by the interconnect driver. In a later study they use a group-based coordination algorithm [**103**]. This algorithm is similar to the staggered algorithm presented by Vaidya [**266**] which is used to minimize the stress on the file system during checkpointing.

Some modifications to the distributed snapshots algorithm involve using grid and tree based mechanisms for collecting markers [**106**]. The goal of these techniques is to reduce the message size and accounting space overhead required in each process as the distributed system grows.

Some coordinated checkpoint protocols rely on a virtualized sockets interface to handle the channel state [**12, 140, 154, 230, 293**]. Typically, this involves the sender logging the message, and then removing the message from the log after receiving a confirmation that the message was delivered correctly on the remote side [**230**]. Alternatively, one could refill the socket buffers by draining the socket buffers to the receiver before a checkpoint then reflecting them off the sender on restart to refill the receivers internal buffers [**12, 154**]. Crak virtualizes the UDP layer to provide message re-transmission, and channel marshaling [**37**]. The Cruz project uses the Zap CRS and modifies the network stack to create a migratible process with a virtual routable address [**140**]. The ZapC project also used Zap, but implemented the network virtualization completely in the socket virtualization

without any protocol restrictions [**154**]. The DejaVu implementation transparently supports the coordinated checkpoint/restart of processes using sockets through virtualization, but required modifications to the MVAPICH library to support InfiniBand communication due to the complexity of the interface and other implementation requirements [**230**]. The DMTCP project uses a distributed snapshots algorithm and socket virtualization to provide coordinated checkpointing [**11, 12**].

In the BCS implementation, the consistent channel state is established due to special properties of the network that allow messages to be sent only at defined intervals [**204**]. Between each of these intervals the channel is in a strongly consistent state therefor any checkpoint generated during this time is globally consistent [**113**].

Other implementations rely on the application to define synchronization points that are globally consistent, sometimes referred to as "safe points". CLIP uses a user-level semi-transparent library where the user must explicitly request a checkpoint, but does not need to identify the state to save [**51**]. Similarly the MPI_Ckpt library provides the same functionality and semantics which has been implemented with an application-level CRS [**114**] and a user-level CRS [**181**]. The CPPC project advances this work by using compiler analysis to automatically determine "safe-points" in the program to insert calls into the MPI_Ckpt library using a pre-compiler [**226**].

Adaptive MPI [**129, 130**] implements migratable virtual MPI processes as user-level threads. Adaptive MPI provides a checkpoint/restart solution closer to application-level checkpointing than system-level checkpointing since it is not transparent to the user-level application. The MPI application must place checkpoint function calls into its code at points when it can guarantee that no messages are being transferred between processes [**128, 291**]. Adaptive MPI then saves the state of the thread to disk as a checkpoint. Adaptive MPI is implemented on top of Charm++ which, in turn, is implemented on top of a native MPI implementation. Being this far removed from the interconnects significantly degrades performance. Additionally, since MPI processes are implemented as user-level threads and share the global address space, Adaptive MPI places additional constraints on the MPI application such as prohibiting the use of global variables.

**4.6. Uncoordinated Protocols.** Uncoordinated checkpoint/restart protocols do not coordinate the checkpoints between processes and instead use specialized algorithms to determine the set of consistent checkpoints on restart. They do not suffer the synchronization overhead during failure-free operation as with coordinated protocols, but at the cost of complex recovery algorithms. Uncoordinated checkpoint/restart protocols need to keep most or all of the generated checkpoints on stable storage since it is not known until restart which set of those checkpoints are required for restart. This is in comparison with coordinated checkpoint/restart protocols that require one or two (for 2-phase commit) checkpoints from each process to be stored since recovery is guaranteed to be able to occur from the most recent, stable checkpoint. Checkpoints that are never used for recovery are called *useless checkpoints*.

On restart the independent checkpoints are analyzed to determine the most recent set of independent checkpoints that form a globally consistent snapshot. The use of *Zig-Zag Paths (Z-Paths)* [**191, 192**] and *Rollback-Dependency Graphs (R-graphs)* [**278**] are used to determine the globally consistent states. A *Z-code* [**173**] technique can also be used to determine all sets of consistent global states during a region of time, which can be useful in focusing the search space and for on-line garbage collection of uncoordinated checkpoints. *T-graphs* and *S-graphs* have also been used to find globally transitless states [**126**].

Often uncoordinated checkpoint/restart protocols are used in combination with message logging techniques. The message logging techniques make the recovery algorithm simpler since most or all messages are accounted for in the logs reducing or eliminating the opportunity for orphan processes. When using a pessimistic message logging protocol then the recovery line becomes easy to form, since no orphans are possible, it is the last checkpoint and most recent log from the failed process. For causal and optimal message logging protocols, a maximal recoverable state must be determined from a list of available checkpoints and message logs to ensure that the restarted state is consistent [**145**]. See Section 5 for more discussion of message-logging protocols.

**4.7. Message Induced Protocols.** Message induced (a.k.a. communication induced) checkpoint/restart protocols attempt to combine the positive aspects of coordinated checkpointing (only require the last set of checkpoints) and uncoordinated checkpointing (no failure-free synchronization of checkpoints). *Forced checkpoints* are checkpoints that the protocol requires an application to take before or after receiving a message in order to maintain a globally consistent state. Message induced checkpointing relies on piggybacking dependency information on messages between processes to determine when to take forced checkpoints. Processes are allowed to checkpoint independently, but must piggyback information about this action on all messages in case it causes an inconsistent global state which would require a peer process to take a forced checkpoint [**151, 174**]. To determine whether or not to take a forced checkpoint typically a dependency graph (e.g., Z-path [**191, 192**]) is piggybacked on all messages [**125**].

Sometimes the message dependency tracking is less explicit and only forced checkpoints are generated by analyzing the communication pattern of the application to determine when a globally consistent checkpoint may be taken [**238**]. The challenge is with dynamic communication patterns it is possible that a checkpoint may never be generated.

Analysis of message induced protocols has largely concluded that such protocols generate a large number of forced checkpoints further stressing the storage system [**10**]. Additionally, the amount of data that is required to be piggybacked can incur intolerable overheads in messaging, and coordinated protocols provide a more consistent overhead while guaranteeing the usefulness of all checkpoints generated [**286**]. The storage and message overheads also challenge the assumption that these techniques scale better than coordinated algorithms since they do not require explicit global synchronization [**10**].

## 5. Fault Recovery: Message Logging

Message logging techniques record message events to a log that can be replayed at a later time to recover a failed process from its initial state. Message logging is a sub-domain of event logging focused on recording message state instead of all non-deterministic events

that impact application execution. Message logging is usually combined with checkpointing to allow for the recovery from an intermediary state of execution instead of from the beginning of execution. Message logging techniques fall into one of three broad categories: *pessimistic*, *optimistic*, and *causal*.

All message logging techniques require the application to adhere to the *piecewise deterministic assumption* which states that "...the state of a process is determined by its initial state and by the sequence of messages it delivers" [**222**]. For message logging techniques, this assumption requires the process recovery to be repeatable (deterministic) and message driven (no need to track other events) [**250**]. When using uncoordinated checkpointing and message logging, *orphan* processes become problematic and influence the choice of a message logging strategy. Orphan processes are surviving processes whose state are inconsistent with the recovered state of a failed process [**9**]. When discovered, orphan processes are forced to rollback to the previous checkpoint. The cascading rollback of orphan processes back to the initial state is called the *domino effect* and can negate all of the benefits of an uncoordinated checkpointing technique [**219**]. The *always-no-orphans* property which is critical to proving the stability of both pessimistic and causal message logging protocols is defined in [**9**]. The log must contain information about the ordering, source, destination, sequence number, and data contents of a message in order to correctly replay the message. This tuple is called the message's *determinate*. Some applications of message logging focus separating the ordering information from the contents in the determinate. By just preserving the order, and not the contents, the determinate for these techniques requires that messages are resent on replay [**227**].

*Pessimistic* message logging techniques require a process to synchronously log to stable storage the message before sending it to the recipient or before delivering the message locally on the application [**24, 27, 29, 46, 68, 86, 167, 194, 247**]. These techniques never create orphan processes since all messages are never lost. They do incur the performance penalty of synchronously logging every message to stable storage, in comparison with other techniques.

*Optimistic* message logging techniques asynchronously log messages to stable storage in order to reduce the performance penalty of pessimistic techniques [**145, 193, 250**]. Since there is an opportunity for messages to be lost to the log, optimistic message logging techniques suffer from the possibilities of orphan processes and subsequently the domino effect.

*Causal* message logging techniques attempt to combine the positive aspects of pessimistic (no orphans) and optimistic message (low overhead) logging by piggybacking volatile determinants on messages between processes [**7, 8, 9, 21, 22, 28, 29, 85, 163, 222**]. The cost of causal message logging techniques is the overhead of piggybacking volatile determinants on every message in the system.

*Adaptive* (or hybrid) message logging techniques track dependency information in relation to checkpoints in order to log only those messages that cross a recovery line, and therefore could produce an orphan process [**56, 189, 190, 287**]. This is intended to reduce the stable storage overhead of logging by reducing the number of messages that need to be logged.

A set of optimal message logging parameters have been presented in [**9**]. The number of forced rollbacks should be close to zero, as in pessimistic protocols. The amount of idle time added to a process during failure-free execution should be close to zero, as in optimistic protocols. The number of additional messages should be close to zero, as in optimistic protocols. The additional size of the existing messages should be close to zero, as in pessimistic protocols.

## 6. Fault Recovery: Replication

Process replication keeps multiple copies (or *replicas*) of a process running and synchronized so that when one process fails a copy can continue execution in its place. A single primary and multiple backup replication technique uses a single process for communication, but keep multiple backup copies synchronized with the state of the interaction [**6, 86**]. In contrast to a primary point of contact, the client can contact all replicas directly using multicast operations [**120**]. Voting and consensus algorithms are often used to determine

acceptance of transactions or values to distribute to external sources [**112, 157**]. The dependence on total ordering of the multicast operation is critical to maintaining consistency among the replicas.

Depending on the style of replication employed, scalability and performance problems may be an issue, leading some researchers to present a hierarchical application of a variety of techniques [**117**]. The degree of fault tolerance depends directly on the number of replicas maintained for each process in the system. A system can only withstand the concurrent loss of one less than the number of replicas of a process since at least one must remain in order to continue computation and possibly recover the lost replicas.

*Roll-forward* recovery techniques use duplex replication to verify program behavior and checkpoints in the case of soft errors during computation [**212, 213**]. In the case of process loss, a checkpoint can be used to restore a process and bring it into a consistent state with the cooperation of its replica peer. Passive replication combined with pessimistic message logging is another way to enable forward recovery [**194**]. *n-Modular Redundancy* is a popular technique to provide high availability in server and multiprocess environments [**15, 16, 44, 89, 110, 162**]. MPI/FT [**16**], VolpexMPI [**162**], P2P-MPI [**110**], and rMPI [**97**] have all attempted to apply replication to the MPI environment.

## 7. Debugging

Debugging has a long history in software engineering [**60, 116, 118**]. Reverse execution or back-stepping allows a debugger to step backwards through the program execution to a previous state, in addition to stepping forwards to the next state. Reverse execution is commonly achieved though the use of checkpointing [**95, 281, 282**], event/message logging [**26, 161, 228, 292**], or a combination of the two techniques [**23, 150, 152, 201, 272**]. When used in combination the parallel debugger restarts the program from a checkpoint and replays the execution up to the breakpoint. A less common implementation technique is the actual execution of the program code in reverse without the use of checkpoints [**4**]. This technique faces challenges of handling complex logical program structures which can interfere with the end user interaction and applicability to certain programs. Though most

of these techniques focus on transparently providing reverse execution, some have also explored language and compiler extensions to support such activities [**289**].

There are three predominate axes to consider when debugging large-scale applications [**152**]. The first axis is that of *runtime*, which becomes a factor when attempting to address a bug that only becomes apparent after hours of computation because of a race condition, or changing computational state in the parallel program. The second axis is the *number of processes* involved, which becomes a significant factor in applications that dynamically adjust the algorithms employed in response to the scale in which the application is run. Finally, the third axis is the *program size* in terms of lines of code involved in the analysis.

Event logging is used to provide a deterministic re-execution of the program while debugging. Often this allows the debugger to reduce the number of processes involved in the debugging operation by simulating their presence through replaying events from the log. This is useful when debugging an application with a large number of processes. Event logging has also been used to allow the user to view a historical trace of program execution that can be inspected while debugging to trace the changes of a variable in reverse without re-execution [**259, 260**].

Message logging is a sub-domain of event logging in which only messages are logged instead of all non-deterministic events that might influence the application. This has been implemented above the MPI interface [**68**] and within the implementation [**26, 176**] for the explicit purpose of supporting debugging operations. There are two core techniques for event replay: Contents-based replay and ordering-based replay [**229**]. In *contents-based* replay, the traces include the values of the events received or variables read. This typically produces larger trace files, but does not require the participation of all processes during replay since values do not need to be recomputed. In *ordering-based* replay, the traces include the relative order in which the events occurred. This produces smaller trace files at the cost of recomputing the values every time. The relative partial ordering of events is based on Lamport clocks [**158, 227**]. Both [**161**] and [**229**] present algorithms for ordering-based replay, whereas [**176**] presents an algorithm for content-based techniques.

Adaptive message logging and checkpointing techniques have also been explored to reduce the size of the message logs [**189, 190, 258, 287**].

Checkpoint/restart is used to return the debugging session to an intermediary point in the program execution without replaying from the beginning of execution. For programs that run for a long period of time before exhibiting a bug, checkpointing can be used to focus the debugging session on a smaller period of time closer to the bug. Checkpoint/restart techniques are also useful for program validation and verification techniques that may be allowed to run concurrently with the parallel program on smaller sections of the execution space [**243**]. [**192**] discusses how to achieve replaying without using message logging by employing a technique similar to message induced checkpointing (see Section 4.7).

In addition to reducing the amount of time and number of processes involved in the debugging process, *program slicing* is often used to reduce the amount of code that needs to be analyzed [**280**]. This is a useful technique when debugging large software systems such as operating systems.

Some of this work has focused on the automatic validation of message passing programs. Since all messages are traced, debuggers can apply algorithms to detect common parallel programming bugs such as race conditions and deadlocks [**65, 188, 243, 287**]. A technique called *flowback analysis* assists debuggers in finding race conditions based on the causal relationship between events [**179**].

For HPC applications, the MPI standard [**178**] has become the *de facto* standard message passing programming interface. Even though some parallel debuggers support MPI applications, there is no official standard interface for the interaction between the parallel debugger and the MPI implementation. However, the MPI implementation community has informally adopted some consistent interfaces and behaviors for such interactions [**58, 115**]. The MPI Forum is discussing including these interactions into a future MPI standard. Chapter 5 will discuss how Open MPI was extended to include a design that supports C/R-enabled parallel debugging.

## 8. Process Migration

Process migration is the ability to move, or migrate, a process from one resource to another. Though often used for fault tolerance there are other ways to apply this technique [**91**]. Process migration plays a central role in proactive fault tolerance [**267, 277**] (sometimes called *reconfiguration*) in which a set of processes are moved away from a predicted system failure either transparently or with the assistance of the application [**241**]. Additionally, process migration may support resource sharing by migrating processes to and from a shared resource on-demand in order to share the resource among a set of processes in a system. *Load balancing* uses process migration to dynamically manage the load on a distributed system moving processes away from heavily loaded machines to lighter loaded machines [**129, 130**].

The performance overhead in process migration is primarily attributed to the storage aspect of the checkpoint overhead involved when migrating a process from one resource to another. In addition to checkpoint overhead, the issue of residual dependencies is also an important consideration. A *residual dependency* is a dependency on the source machine after the process has migrated to the destination machine. For proactive fault tolerance, the technique must be free of residual dependencies since the source machine is likely to fail shortly after the migration. One residual dependency often forgotten about is that of the network state, and the connections between processes. The marshaling of the network state becomes a central issue in checkpointing distributed systems in general, but also specifically in high performance networking in which connectivity information may be dynamic in nature [**235**]. Above all other considerations, the process must be represented accurately during any checkpoint or migration of the process. The numerical stability of the checkpointing algorithm, especially when migrating between heterogeneous machines, has a direct effect on the stability of scientific computation [**142, 172**].

There are a variety of techniques to reduce the checkpoint overhead due to the I/O requirements of moving a process from the source to the destination machine at the cost of checkpoint latency for process migration activities. Many of these techniques harness the

copy-on-write semantics of operating system-level paging. The *eager* technique is the most direct in which the entire process image is transferred to the destination machine before resuming computation [**139, 170, 261**].

A *pre-copy* technique copies the process image to the destination machine while it is still running on the source machine [**257, 277**]. After a significant portion of the memory is copied, the process on the source machine is suspended and the remaining pages of memory are transferred along with control of the execution to the destination machine.

A *lazy* technique transfers just enough of the process state to the destination machine to resume execution, then pages are transferred from the source machine to the destination machine using a demand-driven copy-on-reference approach [**269, 288**]. The core problem with this technique is the residual dependency on the source machine long after the process has been migrated.

A *post-copy* technique builds upon the lazy technique by requiring that all of the memory pages are eventually transferred to the destination machine, giving higher priority to pages immediately referenced by the process running on the destination machine [**224**]. This removes the residual dependency restriction of the lazy technique. However, as with the lazy technique, it can become difficult to determine the minimal amount of state required to start the process running on the destination machine.

A *quasi-asynchronous* migration technique allows non-migrating processes to continue communication while blocking communication to migrating processes [**66**]. This is advantageous over a synchronous migration technique since non-migrating processes are not stopped for the duration of the migration. Additionally, this does not require any residual dependencies as in many asynchronous migration techniques that require a daemon to forward messages in the system.

## 9. File Systems

Processes that frequently interact with files face many difficulties when choosing a fault tolerance strategy. If the processes creates a checkpoint, it must decide how to account for the state of the file system at the point of the checkpoint so that their application can

recover a consistent file state in addition to computation state. Most transparent checkpointing solutions will preserve the file descriptors and seek positions which is adequate for applications that access files in a sequential manner, never delete files, and only append data to files [**111**]. For example, if a process randomly accesses a file for both reading and writing then these techniques do not apply. The checkpoint of the process could also include the entire file, however depending upon the size of the file this may be a prohibitively expensive operation both in terms of disk space used, and checkpoint latency.

Versioned file systems keep multiple revisions of files as backups that are created and retained automatically by the file system [**183**]. A checkpoint/restart service can use this feature to account for the version of the file in the checkpoint generated, and transparently restore that version when restarting the application in cooperation with the file system [**225, 279**]. Compression techniques can also be used to reduce the size of backup files in the file system [**45, 183**]. Recently, it has been shown that by using specialized hardware and file handling techniques checkpoint and restart rollback recovery techniques can be a viable solution for large-scale computation (specifically petascale and exascale machines) [**73, 74**]. This work echos some of the remarks made in an earlier publication regarding the viability of coordinated checkpointing on petascale systems [**84**].

**9.1. Checkpoint Optimized Stable Storage.** Stable storage is a storage device that survives the maximum number of acceptable faults in the system. Typically stable storage is represented by a logically centralized file system such as a Storage Area Network (SAN) or Parallel File System (PFS). The I/O requirements of checkpointing to stable storage account for a significant portion of the overhead in checkpointing. This is usually because as multiple processes attempt to write checkpoint files concurrently to the stable storage device the network becomes congested, and the bandwidth is quickly exhausted.

9.1.1. *Diskless Checkpointing.* Diskless checkpointing removes the physical disk from the stable storage operation replacing it with unused main memory on peer machines [**207**]. This requires that processes do not use all of main memory, and take advantage of fast write

speeds to main memory versus disk based techniques. Though these techniques cannot survive the loss of all processes, they can survive the loss of a subset of the processes by replicating the checkpoint image amongst multiple peers. The replication degree and technique used determines exactly how many failures can be handled by the system and overhead incurred [**105, 223**]. Diskless techniques are often used by ABFT techniques as a way of storing checkpoint information about the computation in progress [**53, 54, 88, 107, 160, 171**]. Skewed checkpointing takes a more dynamic approach to choosing peers to replicate with in order to improve fault coverage without additional replication [**185**].

9.1.2. *Staging, Staggering, and Striping.* The concurrent writing of checkpoint data from multiple processes to stable storage quickly exhausts the bandwidth to stable storage, especially when it is centralized. To address the bandwidth concerns three techniques have been presented in literature: staging, staggering, and striping.

Checkpoint *staging* uses the local memory or disk to quickly write the checkpoint data reducing checkpoint overhead [**264, 266**]. This local checkpoint is then copied to stable storage concurrently while the application continues execution [**36, 45, 205**].

Checkpoint *staggering* reduces contention on the stable storage medium by reducing the number of processes concurrently writing to the stable storage device [**49, 143**]. Usually this is controlled by passing a token around the checkpointing processes that allows them permission to write to the stable storage device.

Checkpoint *striping* builds upon checkpoint staggering by determining how many processes may write to the stable storage medium at a time [**49, 143**]. In the traditional checkpoint staggering approach only one process is allowed to write to stable storage at a time [**264, 266**]. Stable storage devices built from PFS typically use multiple I/O nodes to improve bandwidth to the file system. By adjusting the number of processes concurrently writing, or the stripe, to accommodate the characteristics of the stable storage device checkpoint performance can be further improved [**103**].

9.1.3. *Checkpoint File Replication.* Replicating checkpoint files in the stable storage medium improves the stability of the medium at the cost of additional copies of the data.

Replication, in the form of RAID [**202**], has proven to be an effective technique when applied to disk based storage systems. Distributed file systems use a set of machines with local storage to form a single logical file system. Distributed file systems typically use RAID techniques to provide the fault tolerance necessary to serve as stable storage mediums [**123, 216, 249**]. Some distributed file systems dynamically manage the placement of replicas in order to improve access time and availability in the distributed file system, using replication both for access time and fault tolerance purposes [**123, 156**]. The management of replication is usually handled by a central broker or distributed using a distributed voting or election algorithm to determine the most recent version of the file [**112**].

RAID-1, or checkpoint mirroring, is a common technique used to replicate checkpoints amongst peers in the system [**5, 36, 40, 67, 76, 185, 205, 242, 291**]. The peers may be processes in the application or dedicated checkpoint servers [**25, 214, 274**].

Incremental versions of checkpoint mirroring have been explored by various file systems, they typically operate on blocks of data within files that are overwritten, which may happen during a checkpoint operation that uses the same file for all checkpoints [**5**]. The replication of blocks of a file instead of the whole file has performance benefits at the cost of a more complex replication maintenance algorithm [**111, 153**].

RAID-5 or parity based techniques (including N+1 parity and Reed-Solomon Coding) are used to withstand one or more concurrent failures depending on the parity encoding technique used [**36, 40, 67, 205, 242**].

Often the replication technique is applied transparently to all files written to the file system. However some file systems allow the user to directly choose the replication strategy per file, and even change this strategy over the lifetime of the file in the file system [**40**].

Combinations of the various techniques forming a hierarchy is a common technique to balance the expense of replication with the fault tolerance coverage requirements [**25, 137, 156, 242**]. The use of overlay networks and lightweight storage mediums have also shown promise as techniques for optimizing checkpoint storage [**195, 196**].

PLFS [**18, 19**] found that by transparently turning an N-to-1 checkpoint write operation (all processes write to a single file) into an N-to-N operation (all processes write to individual files) they were able to significantly improve the overall performance of the checkpoint write and restart read operations at scale. Additionally, the stdchk [**5**] and SCR [**36**] projects showed that by using peer based and staging storage techniques for N-to-N checkpoint write operations they were able to significantly improve the checkpoint write and restart read performance. SCR also reported improvements to overall system reliability as a result of the reduced stress on the PFS [**36**].

## 10. Compiler Based Techniques

Compiler based approaches to checkpoint/restart fault tolerance are typically composed of two components: a pre-compiler, and a runtime support library. The pre-compiler is a source-to-source compiler that augments an existing application with calls into the associated runtime support library in order to provide transparent checkpoint/restart capabilities. Typically the application may also identify potential checkpoint locations, and protected regions of execution, though they are not required to.

To support distributed object migration both the DOME [**17**] and Charm++ [**46**] projects require the application to augment data structures with pack and unpack routines to assist checkpointing and restarting the processes/objects. The compiler then uses these markers to support checkpoint/restart activities during execution. In [**166**] these markers were used to dynamically throttle the checkpoint interval based on time from the last checkpoint and the changing process size.

Systems like Porch [**251**], CATCH [**166**] and the $C^3$ [**33, 35, 175**] compilers place potential checkpoint function calls throughout the code during the pre-compiler stage. These potential checkpoints are activated by the runtime library in accordance with the predefined checkpoint interval, and other system and application metrics. The system presented in [**148**] supports heterogeneous checkpointing of sequential programs.

The IMPACT compiler focuses on recovery from transient processor failure instead of entire process failure [**165**]. The compiler allows the application to adjust the sliding window of instructions that can be retried upon processor failure. This provides a dynamic, compiler based alternative to hardware delay write buffers, with seemingly comparable performance.

# 3

# Checkpoint/Restart Infrastructure

Checkpoint/Restart (C/R) rollback recovery is a technique used to reduce the amount of computation lost to process failure by restoring processes from a previously established point in the computation. Distributed C/R techniques rely on various coordination protocols to produce consistently recoverable parallel application states. When designing a C/R system it is important to first identify the set of capabilities that compose to define such an infrastructure. This chapter identifies the C/R capabilities that were composed to produce a fully coordinated C/R infrastructure, realized in the Open MPI implementation. We do not claim that these seven capabilities are a minimal or maximal set since future research may find that finer-grained or additional capabilities are necessary in order to adapt to future systems.

Seven C/R capabilities were distilled from experimentation, a review of previous C/R-enabled Message Passing Interface (MPI) implementations, and related literature. Six capabilities are detailed in this chapter with the seventh capability, the recovery service, explored in Chapter 4. Below is a summary of the C/R capabilities in the order they are presented:

- Checkpoint/Restart Service (CRS): The interface to the single process C/R system provided by or for the system in order to capture an image of a running process for later recovery.
- Checkpoint/Restart Coordination Protocol (CRCP): The C/R coordination protocol implementation that marshals the network state to guarantee a consistently recoverable distributed state upon restart [**48**].
- Interlayer Notification Callback (INC): Notifying and coordinating subsystems of the MPI implementation around various checkpoint related activities (e.g., checkpoint, restart, migration).
- Stable Storage (SStore): A logical stable storage device abstraction encapsulating where and when local snapshots are stored in the distributed environment to form a global snapshot.
- File Management (FileM): The movement of snapshot related files and directories to and from storage devices possibly across file system and node visibility boundaries.
- Snapshot Coordination (SnapC): Checkpoint life-cycle management: distributing the checkpoint request to all participating processes, monitoring their progress, and synchronizing the final local snapshots to a logical stable storage device.
- Error Management and Recovery Policy (ErrMgr): Error reporting and fault recovery management operations including support for preventative actions such as process migration.

Previous C/R-enabled MPI implementations often combined instantiations of the necessary C/R capabilities mentioned above into their implementation. This inseparable design

made it difficult for researchers to explore alternative techniques inside the same implementation. Identifying the C/R capabilities and describing their relation to one another will help C/R fault tolerance researchers design better implementations that are easier to maintain, spur innovation, and remain flexible enough meet the demands of the High Performance Computing (HPC) community. In the Open MPI project, we clearly differentiate the various capabilities in the implementation and still demonstrate full operational support. This indicates that the inseparable design chosen by previous implementations is not required for a correct C/R implementation.

As with most other C/R-enabled MPI implementations, the C/R capabilities and corresponding implementations in Open MPI only support MPI-1 standard functionality. The MPI-1 standard tends to be sufficient for many MPI applications, so we feel comfortable with the application coverage this restriction implies. The Open MPI implementation of the C/R capabilities provides a solid foundation of MPI-1 support, and support for MPI collective routines that are 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 standard, and component advancements (e.g., hardware collectives).

## 1. Open MPI Architecture

Open MPI is an open-source, high-performance, MPI-2 compliant implementation of the MPI standard [**102, 178**]. Open MPI is also dedicated to supporting fault tolerance research and aspires to provide users with a variety of optional, high performance, scalable, (semi-)transparent fault tolerance solutions. We have augmented the Open MPI implementation to provide users with the option of using a coordinated C/R fault tolerance technique. This chapter discusses the various C/R capabilities involved in a such a technique and describes how these capabilities were realized in the Open MPI implementation.

Open MPI is designed around the Modular Component Architecture (MCA) [**245**]. The MCA provides a set of component *frameworks* to which a variety of point-to-point, collective, and other MPI and runtime-related algorithms can be implemented. The MCA allows

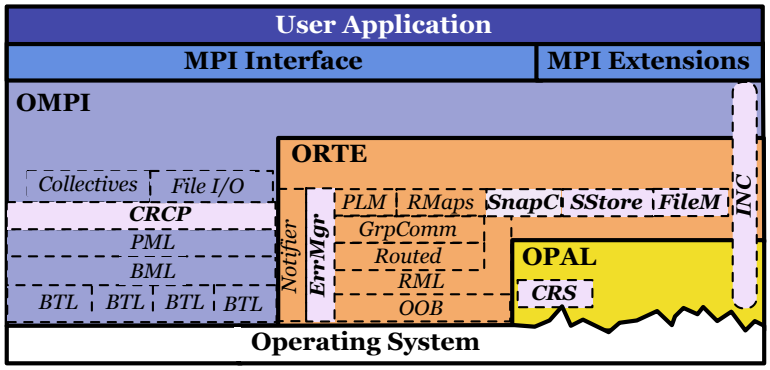



FIGURE 3.1. Illustration of the three layers in Open MPI including some notable MCA frameworks. The highlighted, dashed boxes represent the capabilities discussed in this dissertation.

for runtime selection of the best set of *components* (implementations of the framework interfaces) to properly support an MPI application in execution. By isolating the various C/R capabilities to individual frameworks, researchers can focus on the development of an individual component instead of the development of an entire MPI library implementation in order to experiment with a new technique. Runtime component selection allows researchers to produce a reproducible, accurate comparison of two variants of a capabilities without recompiling the application.

MCA frameworks in Open MPI are divided into three distinct layers: Open Portable Access Layer (OPAL), Open MPI Runtime Environment (ORTE), and Open MPI (OMPI). OPAL is composed of frameworks that are concerned with portability across various operating systems and system configurations along with various software development support utilities (e.g., linked lists). ORTE is composed of frameworks that are concerned with the launching, monitoring, and cleaning up of processes in the HPC environment. The OMPI layer is composed of frameworks that support the MPI interfaces exposed to the application layer. Figure 3.1 illustrates the positions of these three layers.

**1.1. OMPI Layer.** In the OMPI layer, most frameworks (e.g., Collectives, File I/O) sit above the Point-to-Point Management Layer (PML). The PML controls all point-to-point

communication in Open MPI and exposes MPI point-to-point semantics. The PML framework is a stack of three frameworks all working together to provide flexibility and performance to the application. The PML framework breaks messages, with potentially complex data types, into byte streams that can be consumed by the lower layers. The Byte Transfer Layer (BTL) framework encapsulates each of the supported interconnects in Open MPI (e.g., shared memory, Ethernet, InfiniBand, Myrinet MX, etc.). Between the PML and BTL is the BTL Management Layer (BML) that provides the ability to stripe a message across multiple interconnects. This may include using multiple paths between peers, and using multiple interfaces of a single interconnect driver.

The PML exchanges connectivity information, at startup, during a module exchange or *modex* procedure. This connectivity information is used to determine the best possible routes between all peers in the system. This provides the necessary information to all processes in the MPI application so that they can establish communication with any other peer in the MPI application.

**1.2. ORTE Layer.** Many frameworks are combined to form the ORTE layer. The communication frameworks in ORTE combine to provide a resilient, scalable, out-of-band communication path for the runtime environment. The Out-Of-Band (OOB) framework provides low-level interconnect support (currently TCP/IP based). The Runtime Messaging Layer (RML) framework provides a higher-level point-to-point communication interface including basic datatype support. The Group Communication (GrpComm) framework provides group communication, collective-like operations among various processes active in the runtime environment. The Routing Table (Routed) framework provides a scalable routing topology for the GrpComm framework.

The Process Lifecycle Management (PLM) framework is responsible for launching, monitoring, and terminating processes in the runtime environment. The ORTE Daemon Local Launch Subsystem (ODLS) framework provides the same services as the PLM, but on a local node level. The Resource Mapping Subsystem (RMapS) framework is responsible for mapping a set of processes onto the currently available resources.

The ErrMgr framework is accessible throughout the ORTE and OMPI layers. This framework provides a central reporting location for detected or suspected process, or communication failure internal to a process. The Notifier framework works with the ErrMgr framework and provides an interface for processes to send and receive reports on abnormal events in the system, including process failure and communication loss. The Notifier framework also interfaces with external detectors such as the Coordinated Infrastructure for Fault Tolerant Systems (CIFTS) Fault Tolerance Backplane (FTB) [**122**].

In Chapter 4, we will describe how the ErrMgr has been extended to include support for a composable set of recovery policies. Most of the remaining C/R capabilities are realized in the ORTE layer since they do not require access to the MPI layer capabilities, but do require access to the other processes.

**1.3. OPAL Layer.** As with the OMPI and ORTE layers, many frameworks are combined to form the OPAL layer. The OPAL layer is concerned with supporting the current process, and has little to no knowledge of other processes in the execution environment. The OPAL CRS framework provides an Application Programming Interface (API) to the single process C/R service on a particular machine, a fundamental capability in any C/R design.

**1.4. Control Flow.** The C/R infrastructure integrated into Open MPI is defined by process and job levels of control. These two levels of control work in concert to create stable job-level checkpoints (called *global snapshots*) from each individual process-level checkpoint (called *local snapshots*). The various C/R frameworks are represented in Figure 3.2 in their relative position of control.

## 2. External Tools and Interfaces

Defining clear interfaces and abstract tools is important for end user adoption. API interfaces provide the application with direct control from within the program over when C/R related activities are employed and notification on their progress. Command line tools allow the end users to request C/R related activities asynchronously during normal application execution. The *end user* is defined as a person (e.g., software developer, system

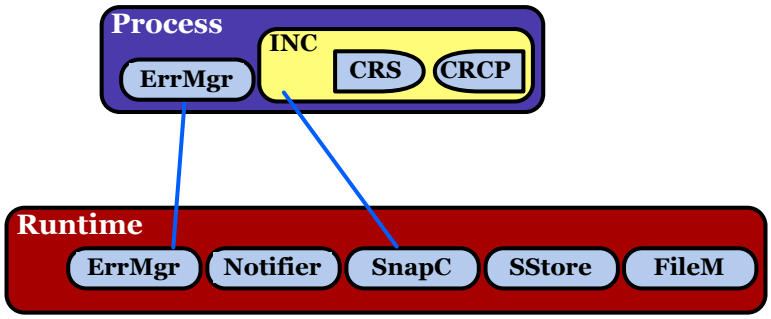



FIGURE 3.2. Organization of Open MPI C/R frameworks in a general MPI application. Job-level control is positioned in the Runtime. Process-level control is positioned in the Process.

administrator, application user) and as an external software agent (e..g, scheduler, resource manager).

**2.1. Application Programming Interface.** The application developer can directly request a checkpoint, restart or migration of the application from within the program using non-MPI-standard APIs provided by Open MPI through the Open MPI Extended Interfaces. These APIs also allow an application to monitor and participate in the C/R application. The specific APIs are described in more detail in Chapter 5 and Appedix A.

**2.2. Command Line Tools.** The end user can asynchronously request a checkpoint of a running MPI application by using the `ompi-checkpoint` command. This command line tool only requires the end user to know the Process Identifier (PID) of the Head Node Process (HNP) process (a.k.a. `mpirun`), and does not require the end user to know any of the internal details of how the checkpoint will occur. This is in contrast to other C/R-enabled MPI implementations which require the end user to track this metadata by hand.

To restart the MPI application using a previously established checkpoint the end user is provided the `ompi-restart` command. The end user passes the global snapshot reference reported by `ompi-checkpoint` to the `ompi-restart` command. The `ompi-restart` command uses the metadata stored at checkpoint time to restart the MPI application.

The `ompi-migrate` tool provides a command line interface for end users to request a process migration within a running MPI application. In addition to allowing the end user

to specify which processes to migrate, this command line interface allows an end user the ability to provide a suggested list of target nodes to use in replacement for the affected nodes. The ability to suggest destination nodes allows a system administrator, for example, to move processes from a set of nodes going down for maintenance to a set of nodes dedicated to the process for the duration of the maintenance activity. This tool also allows end users to experiment with using process migration for load balancing since they can also specify specific process ranks in MPI_COMM_WORLD instead of just nodes for migration.

The interface to these three tools is described in more detail in Appendix B.

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

Previous C/R-enabled MPI implementations required the user to remember specifics regarding exactly how the application was started in order to successfully checkpoint and/or restart the parallel application. A system administrator or scheduler is completely precluded from this runtime information, and must consult the user for these parameters before taking action with a job.

Additionally, these same implementations tended to burden the end user with the responsibility of tracking the location of all the individual checkpoint-generated files. Depending on the CRS, the file set may be one or many files with specific naming conventions. Tracking sets of requests quickly becomes tedious and error prone. This task may be considered an abstraction violation since the end user, to find the restart files, must know the CRS(s) used by the C/R-enabled MPI implementation for the application.

Our design 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: *local* and *global*. The *local snapshot reference* is a single process checkpoint. The local snapshot reference refers to a directory containing a metadata file describing: the CRS used; any application

```
shell$ ls opal_snapshot_0.ckpt
ompi_blcr_context.29121
snapshot_meta.data
shell$ cat opal_snapshot_0.ckpt/snapshot_meta.data
# Timestamp: Thu Apr 01 01:02:03 2010
# PID: 29121
# OPAL CRS Component: blcr
# CONTEXT: ompi_blcr_context.29121
# Timestamp: Thu Apr 01 01:02:04 2010
```

FIGURE 3.3. Local Snapshot Reference Directory Structure and Metadata Contents.

specific parameters; unique checkpoint identifier. The directory also contains all of the single process C/R specific files. Each snapshot generated is designated a unique identifier, usually the rank in MPI_COMM_WORLD, that differentiates one local snapshot from another. Figure 3.3 shows the local snapshot directory structure and metadata contents in Open MPI.

The second type of snapshot reference is the *global snapshot reference* that refers to a collection of local snapshots resulting from a single checkpoint request of an application. 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 in MPI_COMM_WORLD); runtime parameters; a global checkpoint sequence number. The *sequence number* is a monotonically increasing number starting at 0 unique to the job that differentiates subsequent checkpoints from one another. The global snapshot directory also contains the physical set of local snapshots, one from each process in the checkpoint interval. Figure 3.4 shows the global snapshot directory structure and metadata contents in Open MPI.

The global and local snapshot references abstract the user away from the number and name of the checkpoint generated files, alleviating the need for the end users to track multiple files for a single distributed checkpoint interval. The user is only responsible for the

```
shell$ ls ompi_global_snapshot_10587.ckpt
0
global_snapshot_meta.data
shell$ ls ompi_global_snapshot_10587.ckpt/0
opal_snapshot_0.ckpt
opal_snapshot_1.ckpt
opal_snapshot_2.ckpt
opal_snapshot_3.ckpt
shell$ cat global_snapshot_meta.data
# Seq: 0
# Local Snapshot Format Reference: opal_snapshot_%d.ckpt
# Timestamp: Thu Apr 01 01:02:03 2010
# AMCA: ft-enable-cr
# Process: 1659895809.0
# OPAL CRS Component: blcr
# Process: 1659895809.1
# OPAL CRS Component: blcr
# Process: 1659895809.2
# OPAL CRS Component: blcr
# Process: 1659895809.3
# OPAL CRS Component: blcr
# Timestamp: Thu Apr 01 01:02:05 2010
# Finished Seq: 0
```

FIGURE 3.4. Global Snapshot Reference Directory Structure and Metadata Contents.

preservation of a directory containing all the relevant checkpoint information. Additionally, the end user does not need to know the underlying CRS(s) used in order to properly preserve the checkpoint files.

This design also alleviates the need for the end 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.

The level of abstraction provided by the snapshot references allow for the possibility of heterogeneous CRS support. Single process CRSs 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 CRSs into a single global snapshot. Similarly, in homogeneous environments it may be advantageous to use one CRS on a subset of the processes in the job and another on the rest of the processes. The files generated from these distinct CRSs 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.

## 3. Checkpoint/Restart Service

The Checkpoint/Restart Service (CRS) capability captures an image of a single running process (and all associated threads) for later recovery. System-level CRS 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 CRS implementations tend to exist above the operating system making them more portable by sacrificing their ability to directly view some process details without virtualization. Both varieties of CRSs capture a snapshot of a single process on the system and save it to a SStore designated storage device (discussed in Section 6). Many times the CRS is not able to account for the state of objects that exist outside of the process scope such as file system or network interconnect states which may be required for the proper recovery of an application. For further discussion of single process CRSs, see Chapter 2.

In essence, a local CRS is required to provide two tasks: checkpoint and restart. The CRS must allow an implementation to request a checkpoint of a specific process identified by the 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. The CRS must allow an implementation to request a restart of a process on the local machine

provided a local snapshot reference generated by the C/R during a previous checkpoint operation.

The CRS capability is activated by the INC capability from within the process when the system has been sufficiently prepared for the checkpoint. The CRS will interact with the SStore capability to determine where the local snapshot should be written.

**3.1. Implementation.** Open MPI provides a single process CRS framework in the OPAL layer. It is implemented at the OPAL layer since the CRS framework's functionality is limited to a single machine. The OPAL CRS framework provides a consistent API for Open MPI to use internally regardless of underlying CRS API available on a specific machine. Each such system implements a component in the OPAL CRS framework that matches the framework API to the CRS's API.

The CRS 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 the future, additional functionality may be added to the CRS framework to support advanced features such as memory inclusion and exclusion hints for CRSs that support such operations [**209**].

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 C/R framework is a part of the MPI infrastructure and is therefore initialized and finalized within the library.

There currently exist two components of the OPAL CRS framework. The first component is a Berkeley Lab Checkpoint/Restart (BLCR) implementation. BLCR is a system-level transparent CRS that operates as a Linux kernel module. The second component is a SELF CRS component supporting application-level checkpointing by providing the application callbacks upon checkpoint, restart and continue operations. The API for the SELF CRS component is detailed in Appendix C.

## 4. Checkpoint/Restart Coordination Protocol

A snapshot of a process is defined as the state of the process and all connected communication channels [**48**]. The CRCP capability encapsulates the algorithm necessary to marshall the state of the connected communication channels. Local CRSs are unable to account for the state of communication channels as they require both the knowledge of and the ability to coordinate with remote processes. Given this restriction, a distinct capability 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 CRCPs exist and can be generally classified into one of three categories, as detailed in Chapter 2: coordinated, uncoordinated, and communication or message induced. 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.

The MPI implementation must be provided with an API to use internally when interacting with various coordination protocol implementations. Many protocols require the ability to track all point-to-point messages in the system to aid in recovery [**278**]. Other protocols require the ability to piggyback data on outgoing messages, and take action on incoming messages such as taking forced checkpoints [**174**]. Therefore these coordination services need to be provided access to the MPI implementation's 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.

The INC capability (discussed in Section 5) activates the CRCP capability before the CRS. This ensures that the network state is accounted for before the process state is captured.

### 4.1. Implementation.

The Open MPI CRCP framework implements coordination protocols that control for *in-flight messages* [**48**]. The CRCP framework is positioned in the OMPI layer above the PML framework and tracks all messages moving in and out of the

point-to-point stack. The components are provided access to the internal PML framework [**284**] 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 CRCP bkmrk component in Open MPI is an all-to-all bookmark exchange algorithm similar to the one used by LAM/MPI [**234**], except that instead of operating on bytes it operates on entire MPI messages. In this algorithm, processes exchange message totals between all peers on checkpoint, then wait for the totals to equalize. This equilibrium indicates a quiet or quiescent channel and guarantees no in-flight messages. Care is taken in the coordination algorithm to avoid deadlock during the draining process, and to preserve MPI semantics.

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 (part of the INC described in Section 5) involves shutting down interconnect libraries that cannot be checkpointed and reconnecting peers when restarting in new process topologies. 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.

Positioning the coordination algorithm above the point-to-point stack is different than most transparent C/R-enabled MPI implementations. Most C/R-enabled MPI implementations position their coordination protocols in the individual interconnect drivers. By implementing the algorithm in the interconnect driver, the algorithm needs to track bytes being moved through the interface, monitor special behaviors of the device such as Remote Data Memory Access (RDMA) operations, and, for the most part, does not need to worry as much about MPI-level semantics. However, forcing the state to be saved as part of the interconnect driver within the point-to-point stack requires the application to be restarted in a similar process layout. This technique hinders performance by restricting the ability to choose the fastest routes upon restart. Additionally, the coordination algorithm often

becomes muddled with device driver restrictions and requires the driver to provide strict First-In-First-Out (FIFO) ordering of messages.

By lifting the algorithm out of the interconnect driver and placing it above the point-to-point stack we are able to save the point-to-point state at a high enough level to allow for the reconfiguration of the interconnects upon restart. This reconfiguration enables us to adapt to the new process layout to achieve better performance. This is at the cost of a slightly more complex implementation of the coordination algorithm since it must operate on entire messages with MPI semantic restrictions.

**4.2. Interconnect Driver Support.** There are three distinct phases of a checkpoint operation: *pre-checkpoint*, *continue*, and *restart*. In the *pre-checkpoint* phase the process is provided an opportunity to prepare for a requested checkpoint. This involves bringing external resources (e.g., files, network connections) to a stable, checkpointable state. The *continue* phase occurs just after a checkpoint has been taken and allows the process to recover any external resources that it may have stabilized or suspended during the pre-checkpoint phase. The *restart* phase occurs when the process is restarted from stable storage and provides the process with an opportunity to flush caches, and reconstruct any necessary information needed to continue normal operation.

The CRCP framework is positioned above the PML framework and ensures that the lower levels of the point-to-point stack have been drained of all messages coming from and going to this peer process. We can take advantage of this assurance when moving the point-to-point stack through the three phases of checkpointing. It should be noted that we do not require that all processes drain their messages before checkpointing any individual process, only that the individual process taking the checkpoint, at that moment, must be drained of messages. Future work may explore other coordination algorithms that allow for even looser synchrony between processes, such as [**103, 263, 266**].

After the CRCP capability has completed its pre-checkpoint quiescence operation the INC (described in Section 5) may choose to shutdown all active interconnect drivers in order to avoid problematic interactions between them and the CRSs. However, in order to

minimize the failure-free overhead, the INC will want to do only the minimum amount of work necessary to bring the process into a checkpointable state. As a performance optimization, our implementation allows interconnect drivers to indicate if they are *checkpoint friendly*, meaning that they can be safely checkpointed while active with the current CRS active on the system. This optimization is a result of recognizing that shutting down and re-initializing all of the interconnect drivers during a checkpoint operation is often expensive and not always necessary. Since this optimization occurs in the INC and outside of the CRCP, the CRCP remains interconnect agnostic even when this optimization is enabled [**135**].

In the restart phase, Open MPI needs to first clear out the previous set of interconnects and connectivity (i.e., `modex`) information since the machine set and corresponding network address will have typically changed. Then Open MPI re-exchanges the `modex` to get the new connectivity information and reconnect the processes selecting a new set of interconnect drivers (i.e., BTLs) that best match the new process layout.

4.2.1. *TCP/Ethernet Driver.* The Ethernet driver in Open MPI (`tcp` BTL) is classified as checkpoint friendly since it does not need to do anything during the pre-checkpoint phase. This is because the file descriptors of the open sockets are preserved across a checkpoint operation by most CRSs. Since nothing was closed during the pre-checkpoint phase, nothing needs to occur during the continue phase. The restart phase must make sure to close the old socket file descriptors before the `modex` can occur so as not to waste resources.

4.2.2. *Shared Memory Driver.* The shared memory driver in Open MPI (`sm` BTL) is also checkpoint friendly since it also does not need to do anything during the pre-checkpoint phase. This is because the CRS will not checkpoint the contents of open files or shared memory segments, but just preserve the file descriptors to these resources. It should be noted that some CRSs (e.g., BLCR) may, optionally, preserve a shared memory segment shared by processes of the same group (family of processes) as long as they are restarted together. Since the goal of our implementation is flexibility in the process layout on restart, Open MPI does not take advantage of this feature. Instead, we force such CRS to keep only the file descriptor reference to the memory-mapped shared memory file, but not its contents. Upon restart, we must make sure to close the stale file descriptor before the

`modex` operation. The `modex` reestablishes the shared memory segment with the set of peers on the node at restart time which may be different than when originally checkpointed.

4.2.3. *InfiniBand Driver.* The InfiniBand driver in Open MPI (openib BTL) uses the OpenFabrics [**200**] interface to interact with a wide range of InfiniBand hardware. In the pre-checkpoint phase, Open MPI must close the driver which entails closing the ports and releasing all resources allocated on the InfiniBand card, as confirmed by [**104**]. Experimentally, if the BLCR CRS is used with the InfiniBand driver still active, then the state of the kernel becomes unstable and panics which results in node loss. Since Open MPI closes the connections during pre-checkpoint, on continue it must reopen the InfiniBand driver and reestablish connections with its peers. So, because it has to re-exchange the `modex` and reconnect peers, the continue and restart phases look similar at the interconnect level. However, the continue operation can take advantage of the fact that the processes have not changed position, and are likely still reachable via the previously established set of interconnects. So there is no need, for example, to reconnect shared memory segments since processes on the same node have not moved. On restart, Open MPI needs to consider the new topology information when establishing routes in order to account for peer processes may have moved to different machines.

4.2.4. *Myrinet Driver.* The Myrinet driver in Open MPI (mx BTL) uses the Myrinet MX interface to interact with Myrinet interconnect hardware. In the pre-checkpoint phase Open MPI must close the driver, which entails closing the ports and releasing all resources allocated on the Myrinet card; this is similar to the procedure with the InfiniBand driver. However, with the Myrinet driver, the limitation is not kernel instability (as it is with the InfiniBand driver), but reconnecting to the device driver upon restart. On restart, the BLCR CRS will attempt to reallocate the open endpoint file descriptors with the network card, but will receive a permission denied error and fail to restart the application. To avoid this limitation, Open MPI shuts down the mx BTL before a checkpoint. On continue it must reopen the connection to the Myrinet card and reestablish connections with its peers. This requires a `modex` operation to re-exchange the new contact information. Future work may look into ways around this limitation.

## 5. Interlayer Notification Callback

The Interlayer Notification Callback (INC) capability represents the internal coordination necessary to marshallel the many active services within a process to prepare for a checkpoint, continue after a checkpoint or restart a process from a checkpoint. Also referred to as the *pre-checkpoint*, *continue*, and *restart* phases, respectively. This coordination may include flushing caches and activation of C/R specific code paths. The INC provides the infrastructure for process-level control of frameworks to cooperate around a C/R-related operation.

The INC is in charge of activating the CRCP capability to marshall the network state, and the CRS to capture the process state. The INC is managed by SnapC capability which is listening for checkpoint requests in the system.

**5.1. Implementation.** A single process CRS may only preserve a subset of the process state. As previously mentioned, CRSs tend not to account for the state of communication channels. Therefore subsystems within an MPI implementation need to receive notification around C/R requests. In the Open MPI 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 C/R-related activity notification. By attempting to isolate this logic to this function C/R notifications can have a minimal impact upon the implementation of the subsystem making the entire C/R integration more maintainable. The `ft_event` function takes a single state argument indicating the state of the C/R protocol at the time of the function call. The supported states are presented in Table 3.1.

An INC 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 driver notification routines are called Interlayer Notification Callbacks. 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 INC

| State | Description |
|---|---|
| OPAL_CRS_NONE | No checkpoint in progress. |
| OPAL_CRS_CHECKPOINT | Pre-checkpoint phase. |
| OPAL_CRS_CONTINUE | Continue after a checkpoint. |
| OPAL_CRS_RESTART | Restarting from a checkpoint. |
| OPAL_CRS_TERM | Prepare for termination after a checkpoint operation. |
| OPAL_CRS_ERROR | Error has occurred during the checkpoint. |

TABLE 3.1. INC states used in the `ft_event` function.

finishes preparing the library for a checkpoint, it then calls the single process CRS. Once the checkpoint has completed then it uses the `ft_event` function to notify the subsystems of the resulting state of the process.

The design presented so far provides the MPI library the opportunity to prepare for and respond to C/R requests. The application can be viewed as a layer existing above the MPI library. Since some resilient HPC applications may also desire to be notified of such INC state events Open MPI allows them to register INC callback functions, described in Chapter 5 and Appendix A. Multi-layered MPI libraries can use this mechanism to register their INCs. INCs have a similar function definition as `ft_event`. INC callbacks are of the form:

```
int layer_inc(int state);
```

INCs take the same argument as that passed to the `ft_event` function. 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 INC callback. This responsibility ensures a stack-like ordering of INC calls, and gives an INC callback the opportunity to take action before and after calling the previous INC callback which is often from a lower level in the software hierarchy.

In Open MPI each process in the parallel job often 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 runtime environment via the

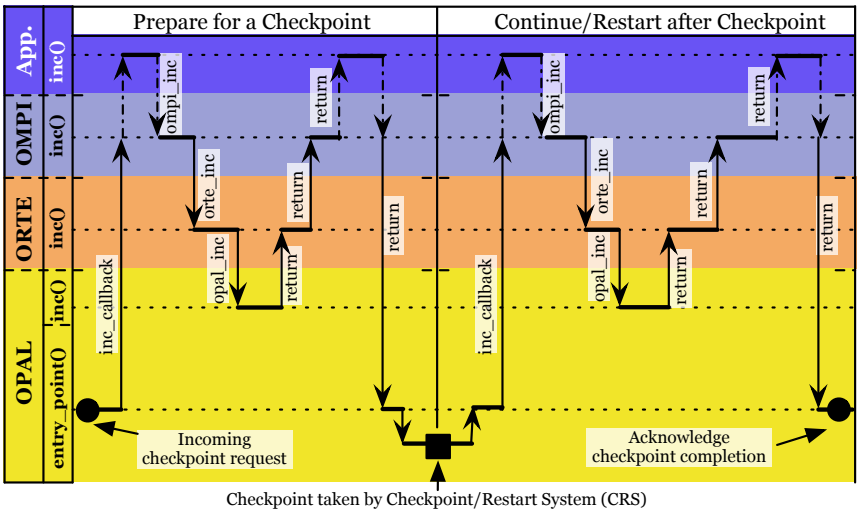



FIGURE 3.5. Illustration of Open MPI Handling a Checkpoint Request. Includes the optional application layer INC callback.

SnapC framework and proceeds into the OPAL `entry_point` function to begin the notification process, as seen in Figure 3.5. If the C/R notification thread is not active then the C/R request is processed when the application process enters or leaves the MPI library. The OPAL `entry_point` function then calls the top most registered INC function. There are three INC functions in Open MPI, one for each layer in the software stack. 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. The API provided to MPI applications for INC registration is described in Chapter 5 and Appendix A.

Since the checkpoint notification thread executes concurrently with other threads in the process, the notification procedure 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, therefore improving maintainability.

## 6. Stable Storage

A *stable storage* device is defined as a storage medium that ensures that the recovery information persists through the tolerated failures and their corresponding recoveries [**82**]. The Stable Storage (SStore) capability encapsulates the technique required to provide a logical stable storage abstraction. In practice, non-transient failure of one or more machines in the system needs to be tolerated by the SStore. Therefore many administrators provide a shared Redundant Array of Independent Disks (RAID) file system that persists past the failure of any machine in the system.

A global snapshot is said to be *established* on stable storage when the snapshot is able to be used to recover the application from stable storage up to the number of anticipated failures in the system. For example, in a centralized stable storage environment (e.g., SAN) that is able to handle the loss of the entire job, the global snapshot is established when all of the local snapshots have successfully been written to the centralized stable storage environment. In a peer-based, node-local stable storage medium that replicates local snapshots among $N$ peers, the global snapshot is established upon verified completion of the replication stage.

The SStore capability uses the FileM capability, as necessary, to move files and directories between storage devices. The SStore capability abstracts the SnapC and CRS capabilities away from the underlying mechanism of how snapshots are established on the stable storage device.

**6.1. Implementation.** The SStore framework in Open MPI currently supports two components: `central` and `stage`. The `central` component stores local snapshots directly to the logically centralized stable storage device (e.g., SAN, parallel file system). The application is stalled until all of the local snapshots have been established to the storage device. If the application was not stalled then a fast process could start messaging a slower process potentially corrupting the checkpoint being generated. The `stage` component uses node-local

storage (e.g., local disk, RAM disk) as a staging location for moving local snapshots back to the logically centralized stable storage device. The application is allowed to continue execution once all of the local snapshots have been written to the node-local storage devices. The ORTE daemon then concurrently moves the local snapshots back to the logically centralized stable storage device, overlapping snapshot movement with application execution often improving the checkpoint overhead.

To improve the performance of C/R based automatic recovery and process migration (discussed in Chapter 4), Open MPI has also implemented local snapshot *caching* and *compression* in the stage component. Local snapshot *caching* requires processes to keep a copy of the last *N* local snapshots (default 2) on node-local storage. This improves the performance of automatic recovery since all of the non-failed processes are not moved in the system, and can use the locally cached copy of the snapshot instead of going to the stable storage device.

Local snapshot *compression* adds one more step in the staging pipeline. After the application writes the local snapshot to node-local storage it is able to continue execution. Once all of the process on the node have finished checkpointing the ORTE daemon compresses the local snapshot using one of a variety of compression utilities (e.g., `bzip`, `gzip`, `zlib`). After the compression stage, the compressed local snapshot is moved to the logically centralized stable storage device. Future work may add support for other checkpoint-specific stable storage file systems (e.g., stdchk [**5**]) and additional compression algorithms.

## 7. File Management

The FileM capability represents the mechanism used to move local snapshots to and from storage devices possibly across file system and node visibility boundaries. Remote file management enables the runtime system to preload files or binaries on remote systems before starting remote processes providing usability conveniences in addition to stable storage abstractions.

The FileM capability 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 determined by SStore. The *remove* operation allows for cleanup of temporary checkpoint data that was preloaded on a remote machine.

**7.1. Implementation.** 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 Open MPI FileM framework interface allows multiple file management requests to be given to the file management system at the same time. This interface also allows a FileM component to potentially use collective algorithms to optimize the operation.

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, local file names, 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 ORTE FileM framework currently uses RSH/SSH remote execution and copy commands (i.e., `rsh/ssh`, `rcp/scp`) to perform the necessary operations. Future work may add components to support standard UNIX commands and high performance out-of-band communication channels.

## 8. Snapshot Coordination

The Snapshot Coordination (SnapC) capability manages a job checkpoint request in the system. It provides job-level control of the C/R capabilities and process-level control over the activation of the INC capability. The SnapC capability controls the checkpoint life-cycle, from distributing the checkpoint request to all processes, to monitoring their progress, to synchronizing the final snapshots to stable storage in cooperation with the SStore capability. Implementations of the SnapC capability support distributed C/R by assuming responsibility for the following tasks upon receiving distributed C/R requests:

(1) Initiate the per process local checkpoints;

(2) Monitor the progress of the global checkpoint operation;

(3) Aggregate the local snapshots into a global snapshot, and

(4) Synchronize the various snapshots to stable storage.

Each C/R-enabled MPI implementation handles these tasks differently depending upon their C/R needs and infrastructure restrictions. The techniques used tend to be tightly integrated into the system, so much so that it is impossible to separate the two.

Implementations of the SnapC C/R capability 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.

The SnapC capability should allow processes 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 SnapC implementation 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 end user should be notified and no processes participating in the request should be affected.

**8.1. Implementation.** Open MPI provides the ORTE SnapC framework to compartmentalize these techniques into components with a common framework API. This compartmentalization allows for a side-by-side comparison of these techniques in a constant runtime environment. The initial ORTE SnapC component, full, implements a centralized coordination approach. It involves three sub-coordinators: a *global coordinator*, a set of *local coordinators* and a set of *application coordinators*. Each sub-coordinator is positioned differently in the runtime environment as shown in Figure 3.6. Figure 3.7 provides a detailed diagram of how the various C/R frameworks in Open MPI work together to take a checkpoint of a running MPI application.

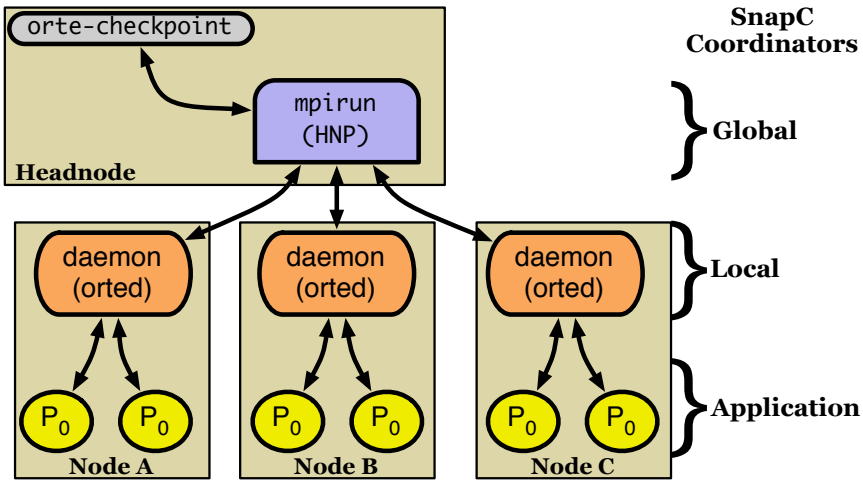



FIGURE 3.6. Illustration of the global, local, and application SnapC coordinators in Open MPI.

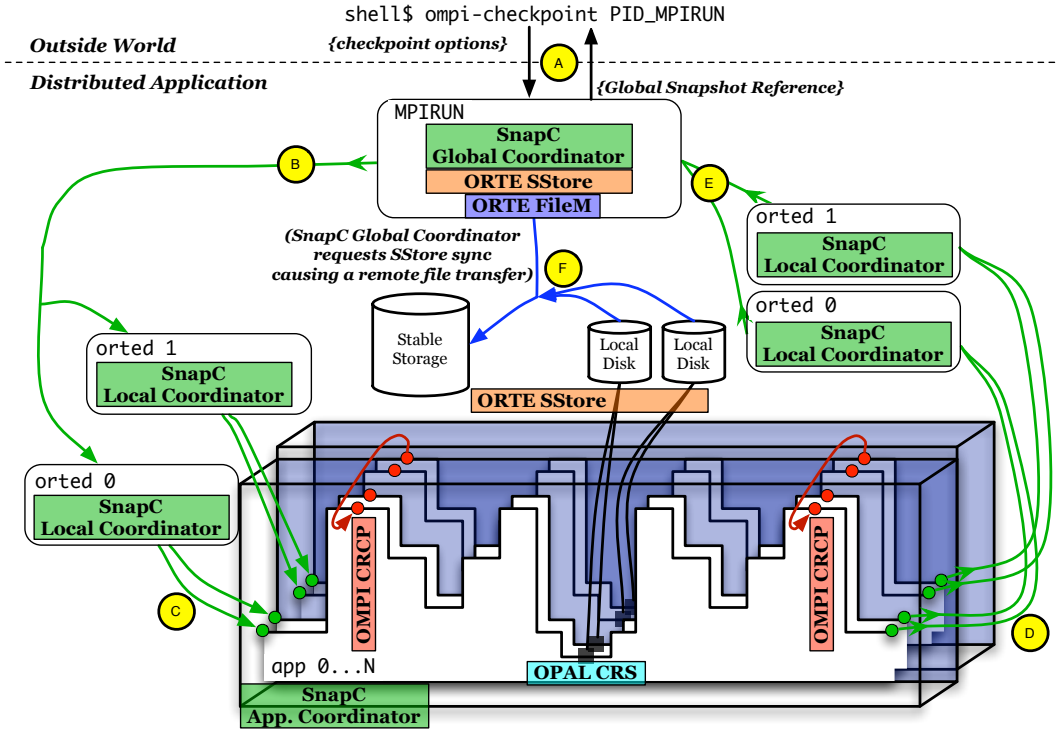



FIGURE 3.7. Illustration of Open MPI C/R frameworks participating in a distributed checkpoint of a running MPI application. 3D boxes represent nodes containing white application processes. Rounded boxes represent runtime support processes.

The *global coordinator* is a part of the HNP (a.k.a `mpirun`) process. It is responsible for interacting with the command line tools (Figure 3.7-A), generating the global snapshot reference, aggregation of the remote files into a global snapshot stored to stable storage using the SStore framework (Figure 3.7-F), and monitoring the progress of the entire checkpoint request (Figures 3.7-B,E).

The *local coordinator* is a part of the ORTE per node daemons (a.k.a. `orted`). Each local coordinator works with the global coordinator to initiate the checkpoint of a single process on their respective machines (Figure 3.7-C), and to move the files back to the global coordinator for storage as a part of the global snapshot in cooperation with the SStore framework (Figure 3.7-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, checkpoint and stop), and calling the OPAL `entry_point` function which starts the INC shown in Figure 3.5.

Once the application coordinator has completed the INC and generates a checkpoint with the CRS, it synchronizes the local snapshot to stable storage using the SStore framework. It then notifies the local coordinator (Figure 3.7-D) that in turn notifies the global coordinator (Figure 3.7-E). The global coordinator then requests the synchronization of the local snapshots to stable storage using the SStore framework (Figure 3.7-F). The SStore framework may use the FileM framework, as necessary, to move local snapshots across file system boundaries. Once these local snapshots have been aggregated and saved to stable storage the global snapshot reference is returned to the user (Figure 3.7-A).

If a failure occurs during the checkpoint operation and there is no recovery option enabled (described in Chapter 4) then the checkpoint sequence number is marked as failed in the metadata by the SnapC and the job terminates. If the SnapC is terminated before it can update the metadata, then the metadata is incomplete for that checkpoint sequence number. On restart incomplete sequence numbers in the metadata are skipped since they

represent invalid or failed checkpoint operations. Only properly finalized checkpoint sequence numbers are used during restart.

## 9. Performance Results

This section explores the performance of the implementation of the various C/R capabilities as realized in Open MPI. This section demonstrates the migration of processes between different network topologies. Additionally, this section looks at the failure-free overhead, checkpoint latency, and checkpoint overhead in this implementation using a variety of micro-benchmarks and real HPC applications.

Experimental results were generated using the Odin and Sif clusters at Indiana University. Sif is an 8 node Dual Intel 1.86 GHz Quad-Core Xeon machine with 16 GB of memory per compute node. It is connected with gigabit Ethernet, InfiniBand SDR, and Myrinet 10G. It is running RedHat Linux 2.6.18-53, BLCR 0.7.3, and a modified version of Open MPI. Odin is an 128 node, Dual AMD 2.0 GHz Dual-Core Opteron machine with 4 GB of memory per compute node. Compute nodes are connected with gigabit Ethernet and InfiniBand SDR. It is running RedHat Linux 2.6.18-53, BLCR 0.8.1, and a modified version of Open MPI.

In this analysis the benchmarks and applications were ran using half of the available cores on each of the compute nodes, as to alleviate some of the memory contention on the nodes. For the compression analysis, the local snapshots were compressed (using `gzip`) on the same node as the application process. Future work may assess the benefits of using an intermediary node to assist in the compression process.

The stable storage overhead discussions primarily used a `noop` program that can be configured with a variable sized random matrix to emulate various process sizes. In these experiments, the `noop` program was given a 10MB random matrix per process.

In the failure-free overhead analysis, we used the NAS Parallel Benchmarks (NAS) (version 2.4) [**187**] to represent a class of typical HPC application kernels. We used 4 of the 8 kernels, specifically the LU class C, EP class D, BT class C, and SP class C kernels. This decision was based on two criteria. First the kernel must run for long enough to perform

the test at 32 or 36 processes on Sif, which meant running for longer than one minute, thus excluding the IS kernel. Secondly, due to storage space restrictions on the SAN, the kernels snapshot image must be less than 15 GB in total size, which excluded the MG class C, FT class D, and CG class D kernels.

We assessed the impact of checkpointing real applications by using GROMACS [**270**], Parallel Ocean Program (POP) [**146**], High-Performance Linpack (HPL) [**203**], and Large-scale Atomic/Molecular Massively Parallel Simulator (LAMMPS) [**211**] HPC software packages. GROMACS is a molecular dynamics application. In our analysis of GROMACS, we used the DPPC benchmark from version 3.0 of their benchmark suite [**270**]. We assessed the impact of SStore configurations, including compression, on the POP, LAMMPS, and HPL HPC software packages. POP is an ocean circulation model [**146**]. In our analysis of POP, we used the *bench01.tacc* benchmark over 5 days of simulation. LAMMPS is a particle dynamics code that supports a wide variety of simulation techniques applicable to biology, chemistry, and material sciences [**211**]. In our analysis of LAMMPS, we used a scaled version of the metal benchmark *eam* involving 11 million atoms over 400 steps. HPL is a popular dense linear algebra benchmark [**203**]. In our analysis of HPL, we used a variety of problem sizes: 40,000 for 64 process, 55,000 for 128 processes, and 70,000 for 192 processes.

**9.1. Network Topology Migration.** To demonstrate the migration of an MPI application between different network topologies we took advantage of the large SMP base of Sif and forced Open MPI to choose certain network drivers simulating different network configurations and availabilities. We used a continuous latency test that measures the time taken for an 8KB message to travel around a ring of 8 processes. This test allows us to account for the shared memory and interconnect latency as if they were a single value. The noise on the Ethernet network seen in Figure 3.8 is caused by administrative traffic on this particular machine.

In our demonstration we checkpointed and terminated the MPI application every 100 steps. Then we restarted it in a different network topology and let it run for another 100

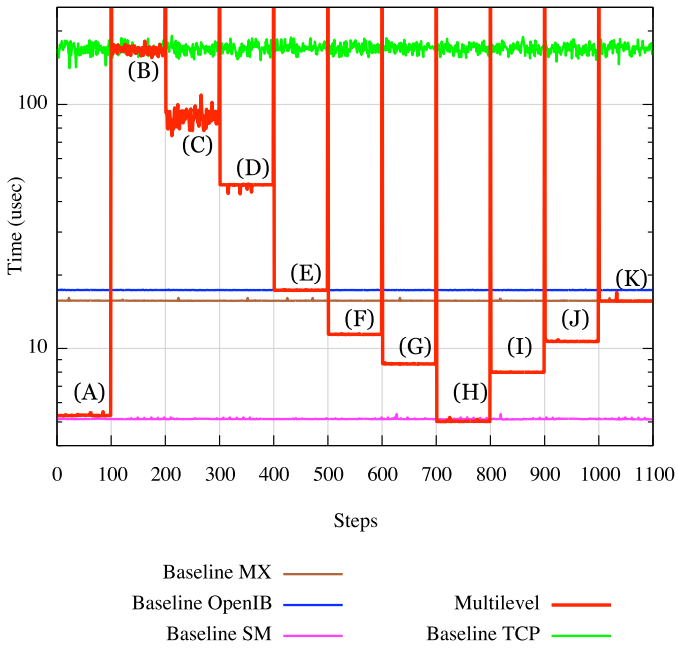



FIGURE 3.8. Continuous latency test with 8 processes exchanging an 8KB message and migrating between different machine configurations. Spikes indicate time spent on disk between the checkpoint and the subsequent restart.

steps before checkpointing and terminating it again. The network topology progression in Figure 3.8 is as follows:

(A) 8 processes on 1 node using shared memory

(B) 8 processes 1 on each of 8 nodes using Ethernet

(C) 8 processes 2 on each of 4 nodes using Ethernet and shared memory

(D) 8 processes 4 on each of 2 nodes using Ethernet and shared memory

(E) 8 processes 1 on each of 8 nodes using InfiniBand

(F) 8 processes 2 on each of 4 nodes using InfiniBand and shared memory

(G) 8 processes 4 on each of 2 nodes using InfiniBand and shared memory

(H) 8 processes on 1 node using shared memory

(I) 8 processes 4 on each of 2 nodes using Myrinet and shared memory

(J) 8 processes 2 on each of 4 nodes using Myrinet and shared memory

(K) 8 processes 1 on each of 8 nodes using Myrinet

(a) NetPIPE 1 byte latency overhead

| Interconnect | No C/R | With C/R | Overhead % | |
|---|---|---|---|---|
| Ethernet (TCP) | 49.92 usec | 50.01 usec | 0.09 usec | (0.2%) |
| InfiniBand | 8.25 usec | 8.78 usec | 0.53 usec | (6.4%) |
| Myrinet MX | 4.23 usec | 4.81 usec | 0.58 usec | (13.7%) |
| Shared Memory | 1.84 usec | 2.15 usec | 0.31 usec | (16.8%) |

(b) NetPIPE bandwidth overhead

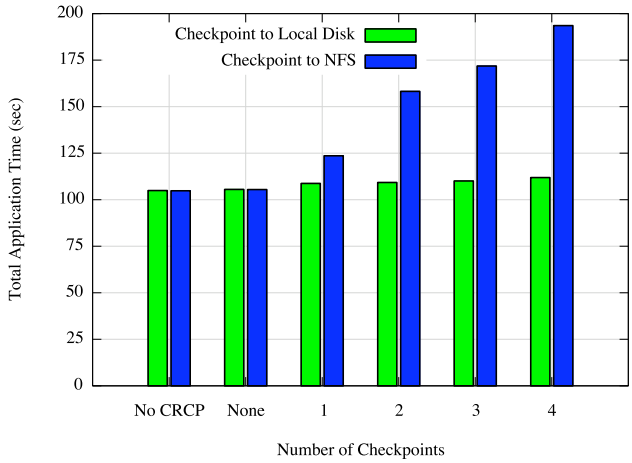
| Interconnect | No C/R | With C/R | Overhead % | |
|---|---|---|---|---|
| Ethernet (TCP) | 738 Mbps | 738 Mbps | 0 Mbps | (0%) |
| InfiniBand | 4703 Mbps | 4703 Mbps | 0 Mbps | (0%) |
| Myrinet MX | 8000 Mbps | 7985 Mbps | 15 Mbps | (0.2%) |
| Shared Memory | 5266 Mbps | 5258 Mbps | 8 Mbps | (0.2%) |

TABLE 3.2. NetPIPE 1 byte latency and bandwidth illustrating CRCP framework failure-free overhead.

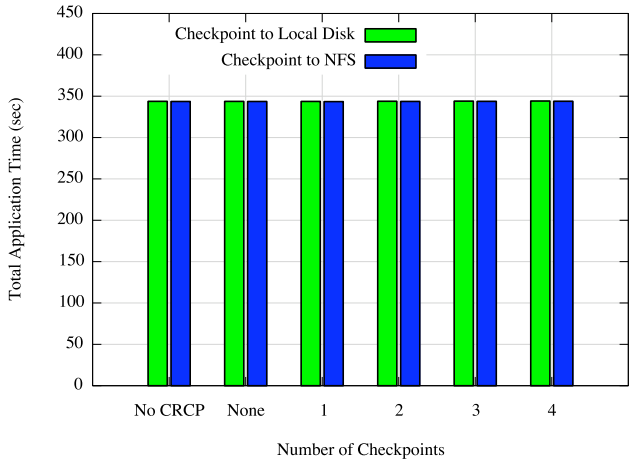
**9.2. Failure-Free Overhead.** Failure-free overhead is the overhead seen by the application during normal operations when a failure does not occur. This overhead includes the time taken by the CRCP protocol monitoring the message traffic, and the additional time to completion for the application when a checkpoint is taken. One of our goals is to minimize the failure-free overhead seen by the application.

One assessment of the failure-free overhead is the overhead seen in the latency and bandwidth parameters between two communicating peers without taking a checkpoint. This measurement accounts for the overhead of the CRCP framework monitoring the message traffic though the system. Using NetPIPE we assessed the latency and bandwidth effects of wrapping the PML layer in Table 3.2. The NetPIPE results show that the differences in bandwidth is negligible, and the 1 byte latency is varies between 0.09 and 0.58 microseconds depending on the interconnect.

**9.3. Checkpoint Overhead.** Another assessment of the failure-free overhead is the performance impact on completion time for an application when various numbers of checkpoints are taken. For this assessment we look at checkpointing to both a globally accessible Network File System (NFS) disk using the central SStore component, and only to the local disk on each machine using a modified stage SStore component that only saves locally. The




(a) LU Class C 32 Processes

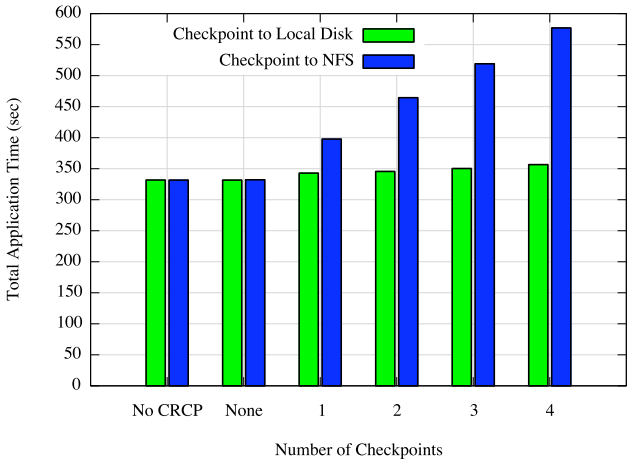



(b) EP Class D 32 Processes

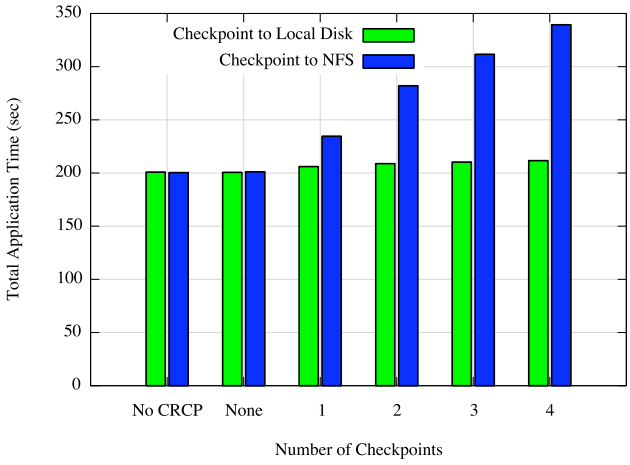
FIGURE 3.9. Performance impact of checkpointing NAS Parallel Benchmarks LU and EP to NFS and local disk.

local disk checkpoint time is provided as a basis for comparison in order to highlight the impact of the file system on the performance of the checkpoint operation.

In Figure 9(a), we look at the effect of checkpointing the NAS Parallel Benchmark LU Class C with 32 processes. The size of the checkpoint is 1 GB or about 32 MB per process. For a single checkpoint, Figure 9(a) indicates that there is an overhead of 17% when checkpointing to NFS and 3% when checkpointing to local disk. For four checkpoints there is an overhead of 84% and 6%, respectively, highlighting the importance of the stable storage




(a) BT Class C 36 Processes




(b) SP Class C 36 Processes

FIGURE 3.10. Performance impact of checkpointing NAS Parallel Benchmarks BT and SP to NFS and local disk.

file system. The checkpoint frequency, or time between checkpoints, for an HPC application is typically measured in hours, in these experiments we are forced to checkpoint more frequently due to the limited runtime of these applications.

In Figure 9(b), we look at the effect of checkpointing the NAS Parallel Benchmark EP Class D with 32 processes. This benchmark creates a checkpoint of 102 MB, or about 3.2 MB per process. Figure 9(b) shows us that the checkpoint overhead is almost negligible.

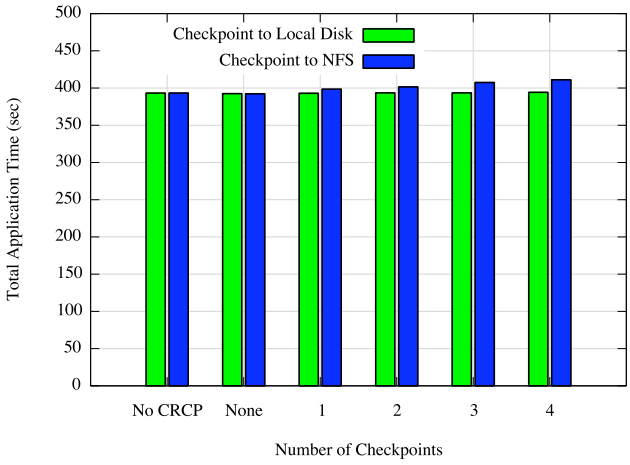
This is due to the small memory footprint and infrequent communication pattern of the benchmark.

In Figure 10(a), we look at the effect of checkpointing the NAS Parallel Benchmark BT Class C with 36 processes. The size of the checkpoint is 4.2 GB or about 120 MB per process. For a single checkpoint, Figure 10(a) indicates an 18% overhead on NFS and 3% overhead on local disk. For four checkpoints there is a 74% and 8% overhead, respectively.

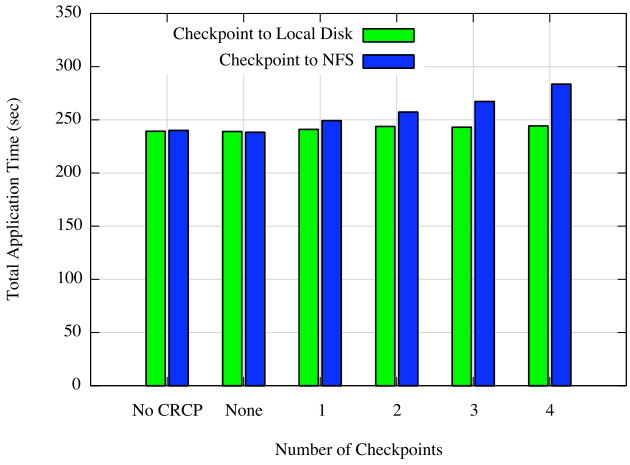
In Figure 10(b), we look at the effect of checkpointing the NAS Parallel Benchmark SP Class C with 36 processes. The size of the checkpoint is 1.9 GB or about 54 MB per process. For a single checkpoint, Figure 10(b) indicates a 17% overhead on NFS and 3% overhead on local disk. For four checkpoints there is a 69% and 6% overhead, respectively.

Next, we assess the performance impact of checkpointing GROMACS with the DPPC benchmark running with 8 and 16 processes. This benchmark creates a checkpoint of 267MB for 8 processes, or about 33MB per process. For 16 processes the checkpoint is 473MB or 30MB per process. The overhead of adding the CRCP layer is negligible, adding at most 1 second to the application runtime for both 8 and 16 processes. Figure 3.11 shows that the performance impact of checkpointing with between one and four checkpoints spaced evenly throughout the execution. The performance impact grows with each checkpoint, but is relatively small for any single checkpoint.

**9.4. Checkpoint Overhead Analysis.** Analyzing the impact of checkpointing on HPC applications is important. It is equally important that the checkpoint overhead is dissected to determine where the checkpoint is spending the most time. In this analysis we split the pre-checkpoint phase into two phases: CRCP Protocol and Suspend BTLs. The former is the Checkpoint/Restart Coordination Protocol (CRCP) and the latter is the time spent suspending interconnect drivers. We expect the time to suspend the Ethernet and shared memory networks to be near zero since no action is taken during the pre-checkpoint phase. Since InfiniBand and Myrinet each have to tear down their network connections, they each are required to spend some time during the suspend phase. The CRCP Protocol time can vary slightly depending on when processes enter into the coordination algorithm and if the




(a) 8 Processes




(b) 16 Processes

FIGURE 3.11. Performance impact of checkpointing 8 and 16 processes with GROMACS DPPC to NFS and local disk.

process is forced to wait on messages to drain from the network. To control for process skew, in this particular analysis, we introduced a barrier between each of the individual operations highlighted in this analysis.

The checkpoint operation is the time it takes BLCR to save the process image to stable storage using the central SStore component. During the continue phase Open MPI may have to rebuild the PML by re-exchanging the `modex`, which is an all-to-all collective operation involving all processes in the application.

| **lu.C.32** | **Ethernet & Shmem** | | **InfiniBand & Shmem** | | **Myrinet & Shmem** | |
|---|---|---|---|---|---|---|
| CRCP Protocol | 0.01 sec | (0.0%) | 0.02 sec | (0.1%) | 0.02 sec | (0.1%) |
| Suspend BTLs | 0.02 sec | (0.1%) | 0.04 sec | (0.2%) | 0.02 sec | (0.1%) |
| Checkpoint | 17.00 sec | (99.9%) | 18.43 sec | (99.3%) | 16.81 sec | (99.2%) |
| Rebuild PML | 0.00 sec | (0%) | 0.08 sec | (0.4%) | 0.10 sec | (0.6%) |
| Total | 17.03 sec | | 18.57 sec | | 16.95 sec | |

TABLE 3.3. Checkpoint overhead analysis for NAS Parallel Benchmark LU Class C with 32 processes using the central SStore component. Global snapshot is 1GB or 32MB per process.

| **ep.D.32** | **Ethernet & Shmem** | | **InfiniBand & Shmem** | | **Myrinet & Shmem** | |
|---|---|---|---|---|---|---|
| CRCP Protocol | 0.10 sec | (8.8%) | 0.09 sec | (3.4%) | 0.09 sec | (5.4%) |
| Suspend BTLs | 0.02 sec | (1.8%) | 0.03 sec | (1.1%) | 0.03 sec | (1.8%) |
| Checkpoint | 1.02 sec | (89.5%) | 2.04 sec | (76.1%) | 1.03 sec | (61.3%) |
| Rebuild PML | 0.00 sec | (0%) | 0.52 sec | (19.4%) | 0.53 sec | (31.5%) |
| Total | 1.14 sec | | 2.68 sec | | 1.68 sec | |

TABLE 3.4. Checkpoint overhead analysis for NAS Parallel Benchmark EP Class D with 32 processes using the central SStore component. Global snapshot is 102MB or 3.2MB per process.

| **bt.C.32** | **Ethernet & Shmem** | | **InfiniBand & Shmem** | | **Myrinet & Shmem** | |
|---|---|---|---|---|---|---|
| CRCP Protocol | 0.04 sec | (0.1%) | 0.07 sec | (0.1%) | 0.06 sec | (0.1%) |
| Suspend BTLs | 0.02 sec | (0.0%) | 0.08 sec | (0.1%) | 0.03 sec | (0.0%) |
| Checkpoint | 67.71 sec | (99.9%) | 68.39 sec | (99.63%) | 69.28 sec | (99.7%) |
| Rebuild PML | 0.00 sec | (0%) | 0.11 sec | (0.2%) | 0.12 sec | (0.2%) |
| Total | 67.76 sec | | 68.65 sec | | 69.49 sec | |

TABLE 3.5. Checkpoint overhead analysis for NAS Parallel Benchmark BT Class C with 36 processes using the central SStore component. Global snapshot is 4.2GB or 120MB per process.

In Table 3.3, we look at the overhead involved when checkpointing the LU Class C NAS parallel benchmark. In Table 3.4, we look at the overhead involved for the EP Class D NAS parallel benchmark. In Table 3.5 and Table 3.6, we look at the overhead involved when checkpointing the BT and SP Class C NAS parallel benchmarks, respectively. For all of these

| **sp.C.32** | **Ethernet & Shmem** | | **InfiniBand & Shmem** | | **Myrinet & Shmem** | |
|---|---|---|---|---|---|---|
| CRCP Protocol | 0.02 sec | (0.1%) | 0.03 sec | (0.1%) | 0.12 sec | (0.4%) |
| Suspend BTLs | 0.02 sec | (0.1%) | 0.08 sec | (0.3%) | 0.03 sec | (0.1%) |
| Checkpoint | 29.76 sec | (98.9%) | 33.02 sec | (99.4%) | 32.19 sec | (99.2%) |
| Rebuild PML | 0.00 sec | (0%) | 0.09 sec | (0.3%) | 0.12 sec | (0.4%) |
| Total | 29.80 sec | | 33.21 sec | | 32.45 sec | |

TABLE 3.6. Checkpoint overhead analysis for NAS Parallel Benchmark SP Class C with 36 processes using the central SStore component. Global snapshot is 1.9GB or 54MB per process.

| **GROMACS DPPC** | **Ethernet & Shmem** | | **InfiniBand & Shmem** | | **Myrinet & Shmem** | |
|---|---|---|---|---|---|---|
| CRCP Protocol | 0.01 sec | (0.2%) | 0.02 sec | (0.4%) | 0.04 sec | (0.8%) |
| Suspend BTLs | 0.01 sec | (0.2%) | 0.01 sec | (0.2%) | 0.01 sec | (0.2%) |
| Checkpoint | 4.76 sec | (99.6%) | 5.24 sec | (98.4%) | 4.95 sec | (97.8%) |
| Rebuild PML | 0.00 sec | (0%) | 0.05 sec | (1.0%) | 0.06 sec | (1.2%) |
| Total | 4.78 sec | | 5.32 sec | | 5.06 sec | |

TABLE 3.7. Checkpoint overhead analysis for GROMACS DPPC running with 8 processes using the central SStore component. Global snapshot is 267MB or 33MB per process.

| **GROMACS DPPC** | **Ethernet & Shmem** | | **InfiniBand & Shmem** | | **Myrinet & Shmem** | |
|---|---|---|---|---|---|---|
| CRCP Protocol | 0.01 sec | (0.1%) | 0.02 sec | (0.2%) | 0.02 sec | (0.3%) |
| Suspend BTLs | 0.03 sec | (0.4%) | 0.04 sec | (0.5%) | 0.02 sec | (0.3%) |
| Checkpoint | 8.07 sec | (99.5%) | 7.88 sec | (98.4%) | 7.48 sec | (97.9%) |
| Rebuild PML | 0.00 sec | (0%) | 0.07 sec | (0.9%) | 0.12 sec | (1.6%) |
| Total | 8.11 sec | | 8.01 sec | | 7.64 sec | |

TABLE 3.8. Checkpoint overhead analysis for GROMACS DPPC running with 16 processes using the central SStore component. Global snapshot is 473MB or 30MB per process.

benchmarks we can see that the time to create the checkpoint and save it to the central stable storage device dominates the checkpoint time. This is closely followed by the time needed to re-exchange the `modex` during the continue phase. The overhead of the CRCP is a factor, but not quite as severe as the time spent in the checkpoint and `modex` operations.

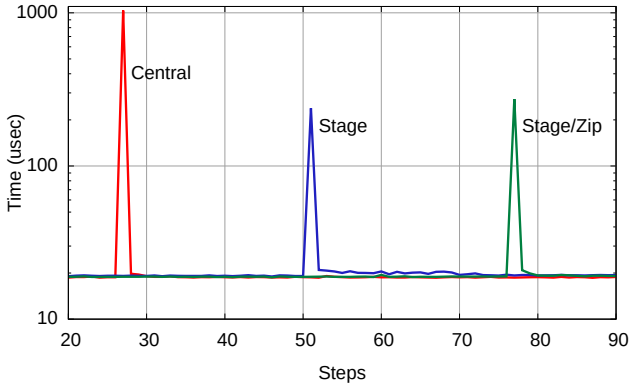



FIGURE 3.12. This illustrates the performance affect of the central, stage, and stage with compression SStore components on application latency. Application latency determined by a continuous latency application using 64 processes exchanging an 8KB message in a ring topology.

In Tables 3.7 and 3.8, we analyze the overhead involved when checkpointing the GROMACS DPPC benchmark with 8 and 16 processes, respectively. This data confirms what was seen with the NAS benchmarks, most notably that the time taken by the CRCP is overshadowed by the time needed to store the snapshots to stable storage.

**9.5. SStore Overhead.** In this section we assess checkpoint overhead by measuring the impact of various stable storage strategies on application performance. We used a continuous latency test that measures the time taken for an 8KB message to travel around a ring of 64 processes on Odin. With this test we are able to both illustrate the impact of the stable storage strategy on the application, and to measure the impact of the checkpoint overhead. Figure 3.12 illustrates the impact of using the following SStore components:

- central
- stage
- stage with compression enabled

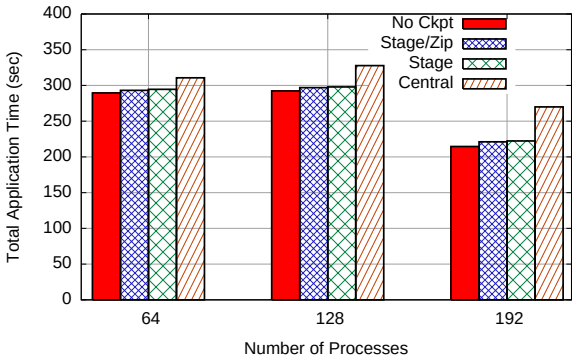
The central SStore component adds approximately 1018 microseconds of half round-trip point-to-point latency overhead across the checkpoint operation which takes 11.1 seconds. The stage SStore component adds approximately 244 microseconds of overhead spread over

21 steps of computation. The checkpoint took 8.3 seconds, spending only 0.8 seconds establishing the local snapshot on the local disk, and 7.4 seconds staging the local snapshots back to stable storage. The compression enabled stage SStore component adds approximately 257 microseconds of overhead spread over 4 steps of computation. The checkpoint took 4.1 seconds, spending only 0.7 seconds establishing the local snapshot, and 2.7 seconds staging the local snapshots back to stable storage.

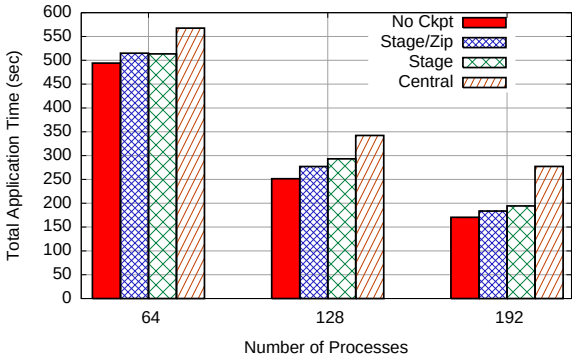
Compression adds slightly more to the checkpoint overhead in comparison with the default stage component, but reduces the duration of the effect. The checkpoint latency is reduced from 8.3 to 4.1 seconds by enabling compression, since this application is highly compressible as it resembles the compression rate of the noop application presented in Table 3.9

The checkpoint latency is reduced from 11.1 to 8.3 seconds by switching from the direct central storage (i.e., central) to the staging protocol (i.e., stage). Often this reduction in checkpoint latency is caused by the flow control in the stage SStore and rsh FileM components which constrains the number of concurrent files in flight to 10, by default. The flow control focuses the write operations so that only a subset of the nodes are using the bandwidth to stable storage at the same time instead of all nodes fighting for the same exhausted bandwidth.

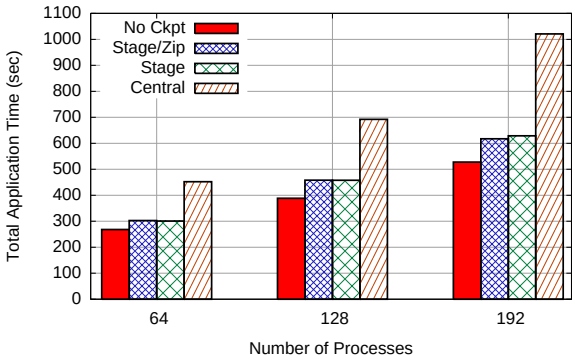
The performance impact on the application runtime for the various SStore component configurations is shown in Figure 13(a) for POP, Figure 13(b) for LAMMPS, and Figure 13(c) for HPL. All of these figures demonstrate that the stage SStore component is an improvement over the central component, especially for large checkpoint sizes. Interestingly, the compression enabled stage component may slightly improve the checkpoint overhead in comparison with the default stage component. Since the compression occurs on-node and competes for computational cycles, one might expect the opposite effect. However, if the application is sufficiently compressible, the overhead involved in checkpointing is regained by reducing the time to establish the checkpoint to stable storage, reducing the overall impact of checkpointing on the network and application.




(a) POP




(b) LAMMPS




(c) HPL

FIGURE 3.13. Checkpoint overhead impact of various SStore components on three HPC applications: POP, LAMMPS, HPL.

**9.6. Compression.** Checkpoint compression can improve checkpoint latency by reducing the amount of data that needs to traverse the network to and from stable storage. The benefits of compression, in terms of improving checkpoint latency, is determined by how

| NP | Application Performance (sec.) | | | | |
|---|---|---|---|---|---|
| | Central | Stage (%) | | Zip (%) | |
| 64 | 0.00 | 0.00 | ( 0.0 %) | 0.00 | ( 0.0 %) |
| 128 | 0.00 | 0.00 | ( 0.0 %) | 0.00 | ( 0.0 %) |
| 192 | 0.00 | 0.00 | ( 0.0 %) | 0.00 | ( 0.0 %) |
| **NP** | **Checkpoint Latency (sec.)** | | | | |
| | **Central** | **Stage (%)** | | **Zip (%)** | |
| 64 | 10.3 | 7.6 | ( 25.8 %) | 4.4 | ( 57.5 %) |
| 128 | 21.8 | 17.7 | ( 18.8 %) | 6.9 | ( 68.4 %) |
| 192 | 40.9 | 28.9 | ( 29.5 %) | 11.7 | ( 71.4 %) |
| **NP** | **Compression Rate (MB)** | | | | |
| | **Normal** | **Zip** | **%** | | |
| 64 | 258.5 | 21.0 | 91.9 % | | |
| 128 | 593.8 | 48.3 | 91.9 % | | |
| 192 | 1167.4 | 93.3 | 92.0 % | | |

TABLE 3.9. Effects of staging and compression on application performance and checkpoint overhead on the noop application.

well the processes address space represented in the local snapshot can be compressed. If the checkpoint does not compress well then this can negate much or all of the benefits of including compression in the staging pipeline. The checkpoint overhead may also increase since, if the compression occurs on the same machine as the process in execution, the two processes could compete for CPU cycles. In order to access the impact of compression on the checkpoint overhead and latency we looked at four applications.

As a baseline number we looked at benchmarking the `noop` program with zero additional bytes of data, effectively a "hello world" style MPI program. Table 3.9 presents the experimental data showing considerable improvements in the checkpoint latency when enabling compression. Since the checkpoint is 92% compressible the checkpoint latency is reduced by 71.4% for a 192 process MPI job. Since the `noop` program waits until it is signaled to finish the Application Performance numbers are not meaningful for this application.

Next we consider the effect of compression on the metal benchmark for the LAMMPS software package. Table 3.10 shows that compression can reduce the size of the LAMMPS global snapshot by up to 67% from 5.3 GB to 1.7 GB reducing not only the checkpoint

| NP | Application Performance (sec.) | | | | |
|---|---|---|---|---|---|
| | Central | Stage (%) | | Zip (%) | |
| 64 | 567.51 | 513.49 | ( 9.5 %) | 514.97 | ( 9.3 %) |
| 128 | 342.29 | 293.13 | ( 14.4 %) | 277.10 | ( 19.0 %) |
| 192 | 277.34 | 194.26 | ( 30.0 %) | 183.67 | ( 33.8 %) |
| **NP** | **Checkpoint Latency (sec.)** | | | | |
| | **Central** | **Stage (%)** | | **Zip (%)** | |
| 64 | 69.1 | 77.3 | (-11.9 %) | 38.0 | ( 45.1 %) |
| 128 | 87.1 | 90.7 | ( -4.1 %) | 36.7 | ( 57.9 %) |
| 192 | 107.1 | 104.1 | ( 2.7 %) | 35.8 | ( 66.6 %) |
| **NP** | **Compression Rate (MB)** | | | | |
| | **Normal** | **Zip** | **%** | | |
| 64 | 4029.4 | 1474.6 | 63.4 % | | |
| 128 | 4638.7 | 1781.8 | 61.6 % | | |
| 192 | 5427.2 | 1786.9 | 67.1 % | | |

TABLE 3.10. Effects of staging and compression on application performance and checkpoint overhead on the LAMMPS application.

| NP | Application Performance (sec.) | | | | |
|---|---|---|---|---|---|
| | Central | Stage (%) | | Zip (%) | |
| 64 | 310.69 | 294.62 | ( 5.2 %) | 293.12 | ( 5.7 %) |
| 128 | 327.74 | 297.88 | ( 9.1 %) | 296.99 | ( 9.4 %) |
| 192 | 269.96 | 222.27 | ( 17.7 %) | 221.10 | ( 18.1 %) |
| **NP** | **Checkpoint Latency (sec.)** | | | | |
| | **Central** | **Stage (%)** | | **Zip (%)** | |
| 64 | 19.7 | 13.8 | ( 29.9 %) | 6.1 | ( 69.0 %) |
| 128 | 35.3 | 27.9 | ( 20.8 %) | 8.5 | ( 75.8 %) |
| 192 | 53.6 | 39.8 | ( 25.9 %) | 14.8 | ( 72.3 %) |
| **NP** | **Compression Rate (MB)** | | | | |
| | **Normal** | **Zip** | **%** | | |
| 64 | 609.3 | 106.2 | 82.6 % | | |
| 128 | 1105.9 | 144.5 | 86.9 % | | |
| 192 | 1802.2 | 205.9 | 88.6 % | | |

TABLE 3.11. Effects of staging and compression on application performance and checkpoint overhead on the POP application.

latency by 67%, but also the amount of stable storage disk space required to store the checkpoint.

| NP | Application Performance (sec.) | | | | |
|---|---|---|---|---|---|
| | Central | Stage (%) | | Zip (%) | |
| 64 | 452.25 | 300.81 | ( 33.5 %) | 302.68 | ( 33.1 %) |
| 128 | 692.32 | 457.59 | ( 33.9 %) | 458.06 | ( 33.8 %) |
| 192 | 1020.76 | 628.66 | ( 38.4 %) | 617.15 | ( 39.5 %) |
| **NP** | **Checkpoint Latency (sec.)** | | | | |
| | **Central** | **Stage (%)** | | **Zip (%)** | |
| 64 | 182.4 | 218.3 | (-19.7 %) | 237.2 | (-30.1 %) |
| 128 | 301.8 | 371.6 | (-23.1 %) | 347.4 | (-15.1 %) |
| 192 | 493.8 | 561.8 | (-13.8 %) | 551.0 | (-11.6 %) |
| **NP** | **Compression Rate (MB)** | | | | |
| | **Normal** | **Zip** | **%** | | |
| 64 | 13557.8 | 13020.2 | 4.0 % | | |
| 128 | 24581.1 | 23234.6 | 5.5 % | | |
| 192 | 40105.0 | 37411.8 | 6.7 % | | |

TABLE 3.12. Effects of staging and compression on application performance and checkpoint overhead on the HPL application.

Next we consider the effect of compression on the *bench01.tacc* benchmark of the POP software package. Table 3.11 shows that compression reduces the size of the global snapshot by up to 89% and the checkpoint latency by up to 76% constituting considerable savings for this application.

Finally we consider the effect of compression on the HPL software package. Table 3.12 shows that compression only reduces the size of the global snapshot by up to 7%. Due to the low compression rate and the large checkpoint size the checkpoint latency increases, but the checkpoint overhead is reduced by up to 40%. All of the application studies show that the checkpoint overhead can be reduced by using a staging approach to stable storage in place of a direct approach.

## 10. Conclusion

This chapter introduced six of the seven C/R capabilities that compose to form a C/R solution for MPI applications on HPC systems. These six capabilities were realized in the

Open MPI implementation of the MPI standard and support basic C/R fault tolerance activities. Using these capabilities we explored the performance implications of C/R on a variety of real HPC applications including HPL, POP, LAMMPS, NAS, and GROMACS.

We demonstrated low failure-free overhead and the ability to restart on different interconnects than when checkpointed. We investigated the performance implications of SStore implementation choices after demonstrating that it can represent up to 98% of the checkpoint overhead. By using a staging SStore technique and compression we were able to improve the checkpoint overhead and latency of our solution for real MPI applications.

# 4

# Process Migration and Automatic Recovery

Resilient applications (i.e., applications that can continue to run despite process failure) depend on resilient runtime and communication environments to sustain them across process failures. In typical HPC environments, communication is provided to the application by an MPI implementation [**178**]. Process launch, monitoring, and cleanup is provided either by the corresponding MPI runtime or by a system-provided runtime. Therefore, a resilient MPI implementation depends on a stable and recoverable runtime environment that can sustain both the MPI implementation and the application. Unfortunately, resilient runtime environments and resilient MPI implementations are uncommon today. As a result, even applications that are designed to be resilient are forcibly removed from the system upon process failure.

The six capabilities presented in Chapter 3 form the foundation of a coordinated C/R infrastructure which provides basic fault tolerance support for MPI applications running on HPC systems. In this chapter we introduce the seventh capability, Error Management and Recovery Policy (ErrMgr), which enables resilient runtime and communication environments to support resilient MPI applications [**134**]. The ErrMgr capability encapsulates the various recovery policies that are supported on the system. The application can then choose an appropriate subset of those policies to support the application's reliability requirements. Ideally, the application will be able to choose more than one process fault recovery policy at runtime to best tailor the recovery policy solution.

We also discuss three currently available recovery policy options implemented as part of Open MPI's ErrMgr framework: *run-through stabilization*, *automatic process recovery*, and *preemptive process migration*. Run-through stabilization supports continuing research into fault tolerant MPI semantics allowing the application to continue running without requiring the recovery of lost processes. Automatic process recovery recovers failed processes in-place using a previously established checkpoint without having to resubmit the job. Preemptive process migration uses the C/R infrastructure to transparently move processes between resources during normal execution.

A logical stable storage device represented by the SStore capability provides recovery policies with a reliable location to place recovery information during normal execution that can be later used during recovery. The implementation of the stable storage device can have considerable impact on both the failure-free and recovery performance overheads of a recovery policy. Accordingly, this chapter analyses the recovery performance tradeoffs for various stable storage strategies including staging, caching, and compression.

## 1. Error Management and Recovery Policy

The Error Management and Recovery Policy (ErrMgr) capability encapsulates the various recovery policies supported on a system. It provides an application with the ability to select a set of process fault recovery policies that best support the application on a given system. Examples of process fault recovery policies include run-through stabilization, proactive

process migration, automatic recovery from a previous checkpoint, and switching to use a replica process. The ErrMgr capability should work with either an external or internal fault notification service. The notification service will inform the ErrMgr of a process failure that it may or may not already be aware of. A good implementation of the ErrMgr capability would allow for the fail-over of one recovery onto another. For example, this would allow for two recovery policies to work together to provide a light-weight and heavy-weight policy. The light-weight recovery policy handles small groups of concurrent failures, and then defer to the heavy-weight recovery policy that can handle a larger groups of concurrent failures.

Individual recovery policies may interact with other capabilities in an implementation. Since the process migration and automatic recovery ErrMgr policies are based in C/R they will interact directly with the SnapC and SStore capabilities to help them migrate and recover lost processes in the system.

### 2. Error Management and Recovery Policy Implementation

In Open MPI, we extended the existing ErrMgr framework to support a variety of recovery policies. Before this extension, the ErrMgr framework provided processes and daemons a stable interface to report process and communication failure so that the proper job abort procedure could be taken. Since recovery is a non-MPI-standard feature, the new ErrMgr framework preserves, by default, the termination of the job upon process failure. We have implemented the following, optional, process fault recovery policies in the new ErrMgr framework: runtime stabilization, automatic recovery, process migration.

The Open MPI ErrMgr framework unites individual recovery policies and techniques explored in previous research [**92, 144**] to support a wide range of policies in a more extensible manner. This unique framework allows policies to be composed in a customizable and interdependent manner so applications can choose from a wide range of recovery policy options instead of just one, as with most previous contributions.

The ErrMgr framework is a *composite* framework that allows more than one recovery policy component to be active at the same time. Active components stack themselves in

priority order and use a *status vector* to communicate actions taken by a component higher in the stack to a component lower in the stack. The status vector is implemented as a bit field that can be inspected and modified in an ErrMgr framework defined manner. If the status vector indicates that no active component was able to recover from a process failure, it activates the job abort procedure. Lighter-weight recovery policies should be ordered higher in the stack than heavier-weight policies. This provides the lighter-weight polices the first opportunity to recover from the failure. If it is able to recover from the process failure, it then sets the appropriate bit in the status vector before passing it down the stack. Lower levels check the status vector before taking any action necessary regarding the process failure. The composable nature of the ErrMgr framework implementation in Open MPI is used by the process migration component to fail-over onto the automatic recovery component when it sees an unexpected failure during a process migration operation. A nice side effect of the composable framework design is that it reduces the recovery policy development burden by supporting and encouraging code reuse.

Fault detection in Open MPI is currently communication and, optionally, heartbeat based. The structure of the detection is determined by the Routed framework. Often the HNP watches the daemons, and the daemons watch the processes on their local machine. If an MPI process detects process failure (i.e., by way of communication timeouts or errors) it reports a *process fault* to the ErrMgr framework. The Notifier provides a generic interface for internal and external event notification. The ErrMgr works in concert with the Notifier framework to propagate the internally detected fault throughout the system. Reciprocally, the Notifier translates external fault events into properly formatted events for the ErrMgr framework. The external fault events could be generated by a system-provided fault monitoring and detection services like the CIFTS FTB [**122**].

The current set of ErrMgr components decide globally how to recover from a failure, so all fault events are forwarded to the ErrMgr components active in the HNP process. The HNP process serves as a *global leader* in the recovery operation since it makes all decisions about the state of the system and how to recover it. It should be noted that even though

the current set of components rely on a global leader, the ErrMgr framework is designed to be general enough to support a more localized or distributed recovery policy.

### 3. Runtime Stabilization ErrMgr Component

The default behavior of the ErrMgr framework is to terminate the entire MPI job upon the detection of process failure. The sustain ErrMgr component provides an application the ability to choose to run-through the process failure by, instead of terminating the job, simply stabilizing the runtime environment and continuing execution.

The Routed, GrpComm, and RML frameworks have been extended to include interfaces for the ErrMgr to notify them of process failure for framework- and component-level stabilization. The individual components are then responsible for recovering from the failure. If the component cannot recover from the loss, indicating an unrecoverable runtime, it can return an error value which will terminate the job.

MPI processes always route ORTE layer out-of-band communication through their local daemon. Therefore, to reduce the per process recovery overhead, the daemon contains the bulk of the recovery logic for re-routing, delaying or dropping communication around recovery from process loss. Dropped communication will return as a communication error to the sending or receiving process.

As part of the runtime stabilization an *up-call* is made available to the OMPI layer from the ErrMgr component to indicate that a process has been lost and that the OMPI layer stabilization and recovery procedures should be activated. Stabilization at the OMPI layer often includes, but is certainly not limited to, flushing communication buffers involving the failed peer(s), activating error handlers and error reporting paths back to the application, and stabilizing communicator and other opaque MPI data structures. The semantics for how MPI functions behave across process failure is still an active area of research and is currently under consideration by the Fault Tolerance Working Group in the MPI Forum [**182**]. The ErrMgr framework was designed to support this effort by providing well-defined stabilization procedures for the runtime environment. The OMPI layer builds upon the stabilized runtime to support research into MPI fault tolerance semantics.

In [**119**], Gropp and Lusk described how a manager/worker style MPI program might recover from process loss by using intercommunicators and forgetting about communicators to lost processes. As an extension to their work, the manager process could decide, after stabilization, to create replacement processes using the dynamic process management (e.g., `MPI_COMM_SPAWN`) interfaces which create intercommunicators between the manager and a new group of worker processes. If the manager is recovering from many failures it will want to overlap the creation of replacement processes with other computation. Since the current dynamic process creation interfaces are all blocking interfaces, the manager must block for each process recovery, or use a separate recovery thread to gain this type of concurrency. We have proposed a nonblocking process creation interface that would allow the manager process to request processes to be created without blocking. The manager can then wait for completion of the process creation requests alongside waiting for completion of normal work units from non-faulty workers. Appendix D describes the nonblocking process management interfaces currently under consideration by the MPI Forum.

## 4. Automatic Recovery ErrMgr Component

Instead of running through a failure, the application may choose to recover from the loss by automatically recovering from the last established global snapshot of the application by enabling the autor ErrMgr component. When autor ErrMgr component is notified of a process failure, by default, it places a failed process on a different node than the one it resided on before the failure. This avoids repeated failures due to node-specific component failure that may have caused the original process failure. Figure 4.1 illustrates a single process recovery in Open MPI.

Since this implementation of automatic recovery is based in a coordinated C/R implementation, all processes must be restarted from a previously established global snapshot in order to provide a consistent state on recovery. The autor component works with the active SnapC component to restart a failed job. Depending on the SStore component active in the application, the local snapshots may be pulled directly from logically centralized stable storage or staged to node-local storage before restart. If there is a locally cached copy of

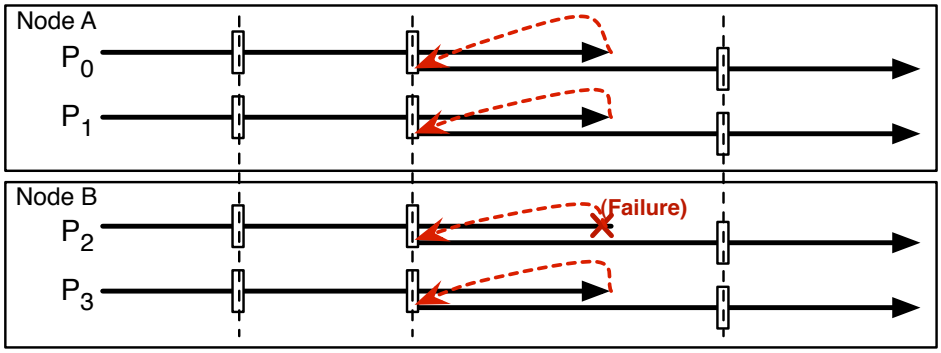



FIGURE 4.1. Four MPI processes running on two machines using the C/R functionality in Open MPI to periodically checkpoint the application (white squares indicate checkpoints). Process 2 fails unexpectedly and all of the processes are automatically, transparently rolled back to their last checkpoint and continue execution using the autor ErrMgr component.

the local snapshot, a process can improve recovery time by using the cached copy to reduce the performance bottleneck in central storage.

If, for some reason, the autor ErrMgr component is not able to recover the job, and no other component is active, then the job aborts. If the stabilize component is active, then the runtime will be stabilized and an error will be returned to the application. This allows an application rely on transparent recovery up to the point it is no longer feasible, and then fall back on an application-involved approach.

## 5. Process Migration ErrMgr Component

When a process or node failure is anticipated, the ErrMgr components are notified via the `predicted_fault()` interface. This interface provides the ErrMgr components with a list of anticipated process and node faults reported by an external fault prediction service or system administrator usually through the Notifier framework. The external fault prediction service can express with each prediction both an assurance level, and an estimated time bound. The estimated time bound allows the process migration ErrMgr component, crmig, to tell if it has enough time to migrate the processes or if it should defer to the autor component for failure recovery. The ability to fail-over on the autor component is provided by the composable design of the ErrMgr framework. The `ompi-migrate` tool provides a

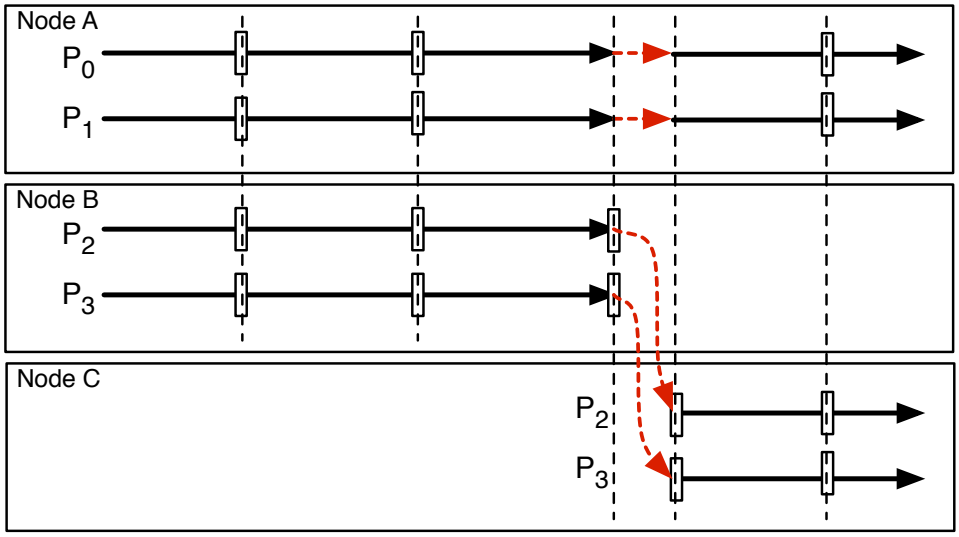



FIGURE 4.2. Four MPI processes running on two machines using the checkpoint/restart functionality in Open MPI to periodically checkpoint the application (white squares indicate checkpoints). The system administrator identifies Node B as going down for maintenance. Open MPI transparently checkpoints the processes running on that machine and migrates them to Node C and continues execution using the crmig ErrMgr component.

command line interface for end users to request a process migration within a running MPI application, described in Chapter 3 and Appendix B. Figure 4.2 illustrates the migration of two processes in Open MPI.

The `ompi-migrate` command line interface allows an end user the opportunity to provide a suggested list of target nodes to use as replacements for the affected nodes. The RMapS framework uses the `suggest_map_targets()` interface to allow the ErrMgr components the opportunity to suggest nodes for each recovering process. The ability to suggest destination nodes allows a system administrator, for example, to move processes from a set of nodes going down for maintenance to a set of nodes dedicated to the process for the duration of the maintenance activity. This tool also allows end users to experiment with using process migration for load balancing since they can also specify specific process ranks in MPI_COMM_WORLD instead of just nodes for migration. The process migration API, provided as part of the Open MPI Extended Interface, allows the application to migrate

processes within a communicator. This API is described in more detail in Chapter 5 and Appendix A.

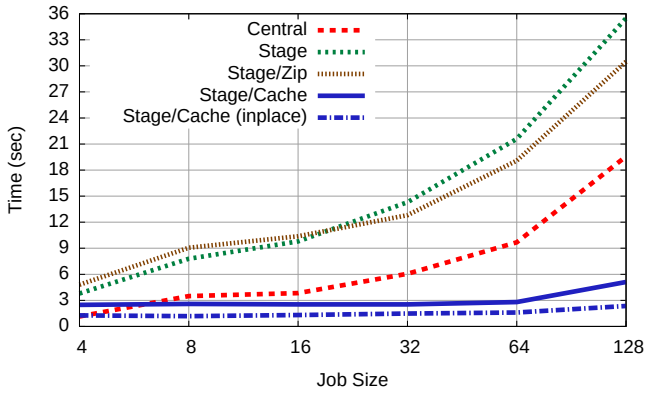
The crmig ErrMgr component implements an *eager copy* process migration protocol without residual dependencies based on the coordinated C/R infrastructure in Open MPI. This component works with the SnapC component to only checkpoint the migrating processes. All other processes are paused after the checkpoint coordination. The migrating processes are restarted on their replacement nodes using the SStore and FileM components as necessary. Then the non-migrating processes are released and computation is allowed to resume. Future work may explore other process migration protocols like *pre-copy* [**257, 277**] and *quasi-asynchronous* [**66**].

If an unexpected failure occurs during process migration, then the migration is canceled and the autor component is allowed to recover the job from the last fully established global snapshot. If the autor component is not active, and no other recovery policy is enabled, then the job will terminate.

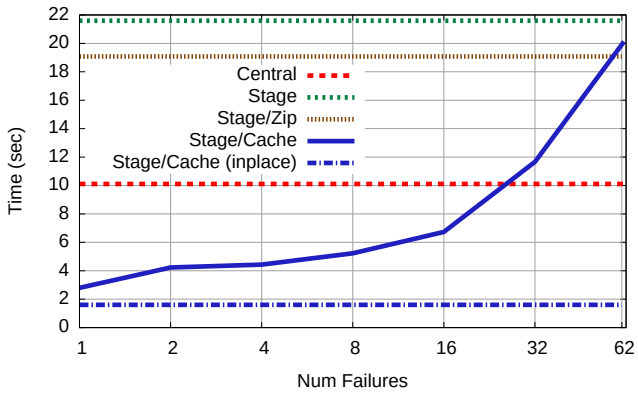
## 6. Performance Results

The experimental setup is the same as in Chapter 3. This section presents an analysis of the affect of stable storage configuration on recovery policy performance. These experiments were conducted on Odin machine at Indiana University.

**6.1. Automatic Recovery.** In this section we assess the performance of the automatic recovery ErrMgr component, autor, for various SStore component configurations. For this assessment, we used the `noop` application to focus our investigation. The `noop` application is a naturally quiescent application (since it does not explicitly communicate) with a fixed process size that allowed us to focus the analysis on the automatic recovery specific overheads. In this experiment failed processes were placed on different nodes than the ones they resided on before the failure. Processes are forcibly terminated by sending `SIGKILL` to the target processes from an external agent.




(a) Impact on a range of job sizes




(b) Impact on a range of concurrent process failures in a 64 process job.

FIGURE 4.3. Performance impact of various SStore component configurations on automatic recovery.

First, we assessed the performance implications of automatically recovering from a single process failure for various job sizes restarting on spare machines in the allocation. Figure 3(a) presents the effect of using the following configurations of SStore components for automatic recovery:

- central
- stage
- stage with compression enabled
- stage with caching enabled

From Figure 3(a), we can see that cache enabled stage component has drastic performance benefits as the job size increases, providing constant a recovery time of approximately 3 seconds. Caching reduces the recovery pressure on stable storage by allowing the non-failed processes to restart from the node-local storage while only the failed processes are forced to stage-in their local snapshots from stable storage.

The central component outperforms the cache enabled stage component for small job sizes. This is because even with a caching enabled stage component the failed processes must copy the local snapshot to the node-local storage before restarting from it. This is in contrast to the central component which avoids the copy to node-local storage by directly referencing the local snapshot. Notice also that the compression enabled stage component begins to outperform the default stage configuration at larger job sizes since it is reducing the amount of data being transferred between the stable storage device and node-local storage.

Figure 3(b) presents checkpoint latency for a variety of concurrent failures in a fixed size job, in this case 64 processes. The time to recover the job is not changed by the number of failures since for all of the non-cache enabled SStore components the entire job is terminated and recovered from stable storage. The main variable in Figure 3(b) is the recovery time when caching is enabled. We can see that caching continues to provide performance benefits up to about half of the job failing, at which point the central component begins to perform better.

Interestingly, even up to 62 concurrent process failures a caching enabled stage component still performs better than the default stage component. This indicates that even with a few processes taking advantage of the node-local cache, in this case two processes, there are still performance benefits to not further stressing the stable storage device.

If we allow failed processes to be restarted on the node in which they previously failed, the benefits of caching becomes even more significant. The time to restart becomes approximately 1.5 seconds regardless of the number of concurrent failures or the job size. This is a slightly unrealistic use case for typical deployments since processes that crash due to node failure often cannot access the original node from which to restart. However, this is

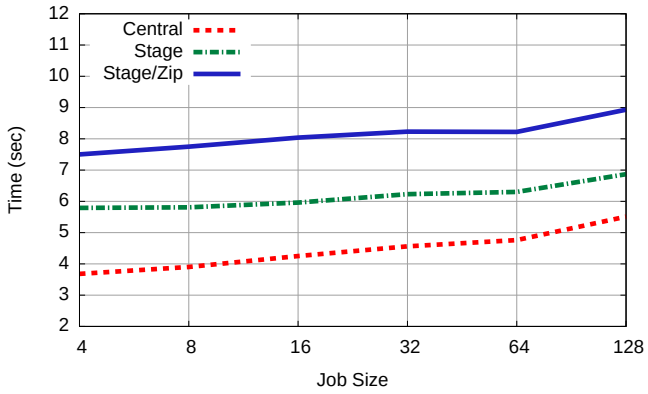
an interesting data point for future investigations into peer-based, node-local storage that eliminates or reduces the need for logically centralized stable storage devices, confirming much of the previous literature [**5, 36**].

**6.2. Process Migration.** In this section we assess the performance of the eager copy process migration ErrMgr component, crmig. As in Section 6.1, we are using the noop application to focus the analysis of the performance overheads involved in process migration. In this experiment, processes were migrated from a source set of machines to a destination set of machines that were distinct from the source set. So in this experiment, caching will not provide any benefit since the source node is never the same as the destination node for any of the migrating processes.

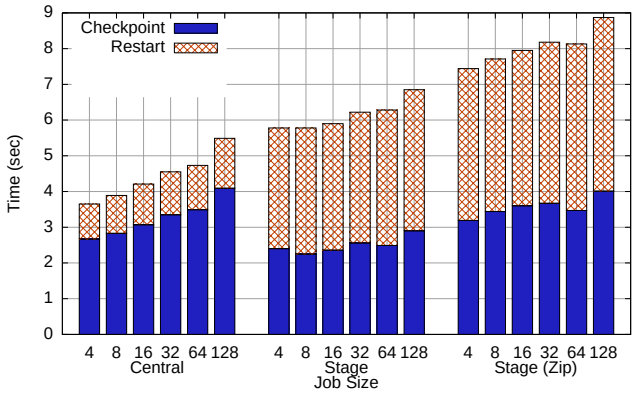
Process migration is often used to move an entire node's worth of processes in anticipation of a node failure. With this use case in mind, we assessed the performance impact of migrating two processes while varying the size of the MPI job.

In Figure 4(a), we can see that the central component performed better than the either of the stage component configurations. Looking at the breakdown in Figure 4(b) we see that the time to restart the processes remains fairly constant regardless of the job size among each SStore component. This is due to the relatively low bandwidth requirement of pulling two local snapshots from stable storage. The significant difference comes in the time to stage the local snapshot to and from stable storage. The reduction in the checkpoint overhead is beneficial when checkpointing for fault recovery, but contributes additional overhead to the process migration performance.

Process migration is limited by the time to move the checkpoint from the source to the destination system. Both the central and stage SStore components rely on a logically centralized stable storage device. As such, they copy the local snapshot to the stable storage device from the source machine then immediately copy it back from stable storage to the destination machine. As future work we may investigate direct copy techniques that remove the logically centralized stable storage device from the process migration procedure. Our discussion in this section focuses on the performance implications of two common stable




(a) Performance impact of SStore components

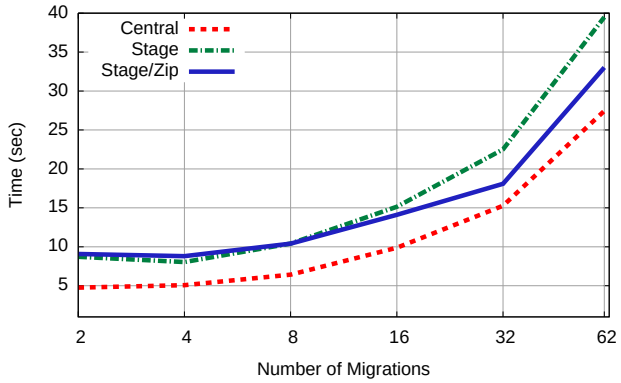



(b) Breakdown of performance impact of SStore components.

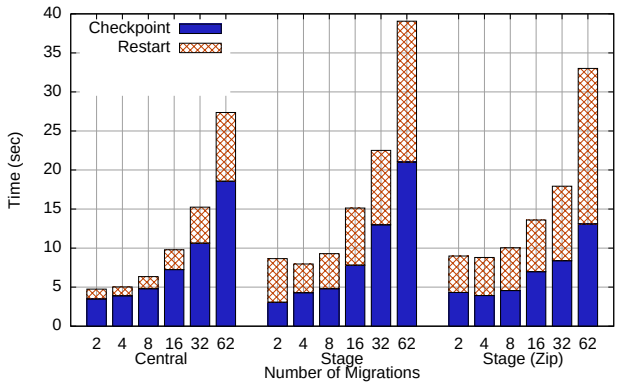
FIGURE 4.4. Performance impact of various SStore component configurations on process migration on a range of job sizes.

storage techniques provided by C/R enabled MPI implementations. So for a small number of migrating processes, the central component requires the least number of copies during the migration, and given the limited bandwidth requirements of migrating only two processes this SStore component is the best performance option.

Next we assessed the performance implications of process migration by varying the number of migrating processes for a fixed size job, in this case 64 processes. Figure 5(a) shows that as the number of processes migrating are increased, the compression enabled stage component begins to outperform the default stage component, and approaches the performance of the central SStore component. If we look at the breakdown of the migration




(a) Performance impact of SStore components




(b) Breakdown of performance impact of SStore components.

FIGURE 4.5. Performance impact of various SStore component configurations on process migration on a range of migrating processes in a 64 process job.

overhead in Figure 5(b), we can see that the time to checkpoint increases as we increase the number of migrating processes, since we are checkpointing more processes and putting more pressure on the bandwidth to stable storage. Reciprocally, we can see the time to restart the migrating processes increases as the number of migrating processes increases. So, it makes sense that the compression enabled stage component begins to approach the performance of the central component since it reduces the amount of data traveling over the network to and from stable storage.

## 7. Conclusion

This chapter presented the seventh and final C/R capability, the Error Management and Recovery Policy (ErrMgr). The ErrMgr encapsulates the recovery policies available in the system. We discussed how the composable design of the ErrMgr implementation in Open MPI allows for recovery policy fail-over. We analyzed the affect of stable storage configurations on recovery performance. We found that caching local snapshots significantly improves the performance of automatic recovery. We highlighted the scalability implications of various SStore solutions and recovery policies.

# 5

# Application Interaction

Previous chapters have focused on presenting capabilities that form a transparent, coordinated C/R solution for MPI applications on HPC systems. Application-transparent techniques can greatly benefit from even minimal application interaction in the form of guidance. For example, applications can identify temporary buffers that can be removed from the checkpoint or highlight regions of program execution where the state is minimal or naturally synchronized. Applications can also benefit by directly interacting with the C/R infrastructure to determine not only when to checkpoint, but also when and where to migrate for fault avoidance or load balancing. In this chapter, we introduce application and debugger APIs that expose the underlying C/R features to the end user. All of the interfaces presented in this chapter are optional for the application, but provide the application

```
// Request a Checkpoint
int OMPI_CR_CHECKPOINT(char **handle, int *seq, MPI_Info info);
// Request a Restart
int OMPI_CR_RESTART(char *handle, int seq, MPI_Info info);
```

FIGURE 5.1. Open MPI Checkpoint and Restart APIs.

with the opportunity to have greater influence on when and how C/R-related activities are executed in the system.

Section 1 focuses on an API for the application. Section 2 focuses on an API for parallel debuggers which operate transparently to the application.

## 1. Application Programming Interface

This section focuses on optional, non-MPI-standard APIs for the application. These are not part of the MPI standard, so are implemented as part of the Open MPI Extended Interfaces. As such they are prefixed with OMPI_CR_. Though many of the interfaces provide an optional `info` argument, no keys are defined at this time except for the migration API. Examples and further details are provided in Appendix A.

**1.1. Checkpoint and Restart Requests.** The application can request a checkpoint of the application by using the OMPI_CR_CHECKPOINT API, seen in Figure 5.1. The checkpoint API allows an application to request a checkpoint when convenient to the application; for example, when the application is in a minimal or naturally synchronized point in time in order to, potentially, reduce the checkpoint overhead.

Additionally, the application can request to restart itself by using the OMPI_CR_RESTART API, also seen in Figure 5.1. The restart interface allows an application to return the computation to a previously established checkpoint in order to work around application-detected faults. For example, this interface is useful for an application that can detect a process behaving in a Byzantine faulty manner, possibly due to soft errors. Once detected, an application could use the restart API to restart the computation from a known, good, previously-established state in the computation before the occurrence of the soft error.

The OMPI_CR_CHECKPOINT returns a *handle* to the global snapshot reference and a unique *sequence number*. Since the C/R implementation in Open MPI is fully coordinated, the checkpoint call is collective over all processes in the MPI application. The OMPI_CR_RESTART function takes as arguments the *handle* and *sequence number* returned by the OMPI_CR_CHECKPOINT or an external tool like `ompi-checkpoint`, described in Chapter 3 and Appendix A.

**1.2. Quiescence Regions.** The quiescence functions allow an application to define a region of the program during which the MPI implementation guarantees that no messages are *in-flight* with regard to the calling process. A message is said to be *in-flight* if it has been sent, but not yet received by the recipient. So the message may be on the sender side, receiver side, or somewhere in between. To have *no messages in-flight* means that all messages that have been sent to this process have been cached inside the MPI implementation, and any messages this process has sent have been cached on the intended receiver side. Or, said another way, any message sent by one process on the specified communicator has been transmitted to the recipient. The recipient may either buffer the message in the MPI implementation (if no receive has been posted) or place it in the recipient's buffer (if a receive has been posted). The MPI implementation can choose how optimally to buffer the message contents (e.g., network card, internal data structure, specialized hardware).

The quiescent region of code is defined between the `start` and `end` functions in Figure 5.2. In this region, the use of the MPI interface is restricted to maintain guarantees about in-flight messages between processes. The MPI process may use any functionality that is local in nature, and post new receives. It may also use the request completion functions. The application may not post any new sends or enter collective operations during this time. This functionality is useful when checkpointing an application, or preparing buffer space for additional communication.

The quiescence functions are collective, so all MPI processes defined in the corresponding communicator must call these functions before any of them can complete. Though the interface allows any communicator to be passed to the functions, Open MPI requires it to

```
// Quiescent Region Start
int OMPI_CR_QUIESCE_START(MPI_Comm comm, MPI_Info info);
// Quiescent Region End
int OMPI_CR_QUIESCE_END(MPI_Comm comm, MPI_Info info);
// Request a checkpoint during a quiescent region.
int OMPI_CR_QUIESCE_CHECKPOINT(MPI_Comm comm, char **handle, int *seq,
                               MPI_Info info);
```

FIGURE 5.2. Open MPI Quiescence APIs.

```
// Migration Request
int OMPI_CR_MIGRATE(MPI_Comm comm, char *hostname, int rank, MPI_Info info)
```

FIGURE 5.3. Open MPI Migration API.

be MPI_COMM_WORLD. The quiescence operations on communicators serve a similar purpose as MPI_WIN_FENCE does to one-sided RMA operations on a window. This collective operation allows the application to force all outstanding communication to the intended recipient of the communication.

The OMPI_CR_QUIESCENCE_CHECKPOINT function allows the application to optionally choose to use the MPI provided CRS. This interface assumes that the MPI library has been quiesced by a previous call to the quiescence start function, in contrast to the OMPI_CR_CHECKPOINT presented in Section 1.1.

**1.3. Migration.** The application can migrate a subset of the processes in a job by using the OMPI_CR_MIGRATE command, depicted in Figure 5.3. The application may want to do so between different phases of computation to better load balance the processes. The group of migrating processes is defined by the communicator passed to the migration function. The application can optionally specify a hostname or rank to move the current rank onto. Future implementations may also allow processes to be moved *close to* other ranks, for some definition of *close to*. The application can set the `CR_OFF_NODE` MPI_INFO keyword to `true` in order to indicate that they wish to migrate off of the current node. In order to reduce the time required to migrate, the mapping algorithm should attempt to reduce the number of processes that have to be moved in order to satisfy the migration request.

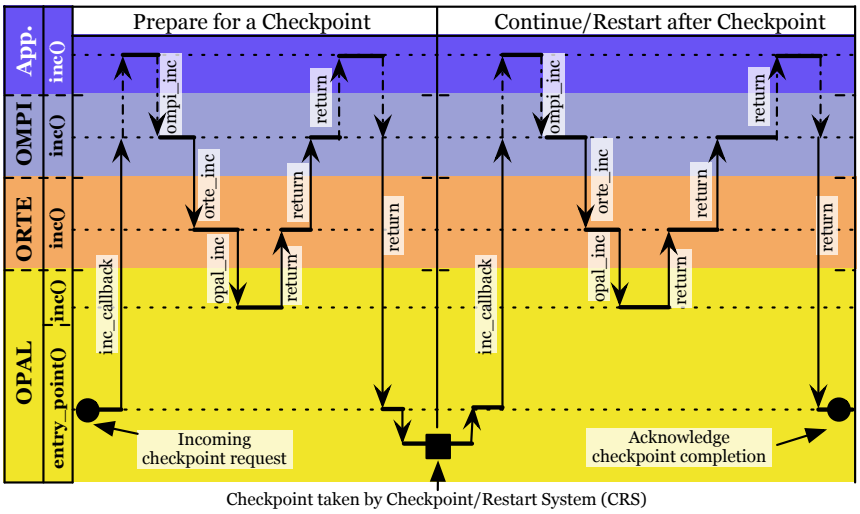



FIGURE 5.4. Illustration of Open MPI handling a checkpoint request with the application involved in the INC.

**1.4. INC Callbacks.** As mentioned in Chapter 3, the application can choose to register a callback function to be notified of C/R-related activities during the INC coordination procedure. The application may use this as an opportunity to synchronize additional libraries (e.g., accelerators [**253**]) or files that would not otherwise be accounted for in the local snapshot. Figure 5.4 shows how the application fits into the INC coordination procedure. The application is provided the first and last opportunity to take action in all three phases of C/R traversed by the INC. By registering a callback function, the application assumes the responsibility for calling the previous callback function, namely `ompi_inc`, which is returned by the registration function. Figure 5.5 presents the function signature and registration functions for INC callbacks. Table 5.1 shows the various parameters used when registering and de-registering callbacks.

**1.5.** self **Checkpoint/Restart Service (CRS).** Though much of the focus in this document has been on using the BLCR CRS for transparent single process checkpointing, Open MPI also provides an application-level CRS called self. self provides callbacks into the application for checkpoint, restart and continue operations when it would normally be called in the INC (shown in the middle of Figure 5.4). These callbacks allow the application to write

```
// INC Registration Function
int OMPI_CR_INC_register_callback(OMPI_CR_INC_callback_event_t event,
                                  OMPI_CR_INC_callback_function function,
                                  OMPI_CR_INC_callback_function *prev_function);
// INC Callback Function Signature
typedef int (*OMPI_CR_INC_callback_function)(OMPI_CR_INC_callback_event_t event,
                                             OMPI_CR_INC_callback_state_t state);
```

FIGURE 5.5. Open MPI INC Registration API.

| **OMPI_CR_INC_callback_event_t** | |
|---|---|
| **State** | **Description** |
| OMPI_CR_INC_PRE_CRS_PRE_MPI | Pre-checkpoint, before OMPI INC. |
| OMPI_CR_INC_PRE_CRS_POST_MPI | Pre-checkpoint, after OMPI INC. |
| OMPI_CR_INC_POST_CRS_PRE_MPI | Continue/Restart, before OMPI INC. |
| OMPI_CR_INC_POST_CRS_POST_MPI | Continue/Restart, after OMPI INC. |
| **OMPI_CR_INC_callback_state_t** | |
| **State** | **Description** |
| OMPI_CR_INC_STATE_PREPARE | Pre-checkpoint |
| OMPI_CR_INC_STATE_CONTINUE | Continue |
| OMPI_CR_INC_STATE_RESTART | Restart |
| OMPI_CR_INC_STATE_ERROR | Error |

TABLE 5.1. Open MPI INC function callback events and states.

their checkpoint into a *black box* directory that is stored with the local snapshot to stable storage as defined by the active SStore component.

Figure 5.6 presents the three default functions that the self CRS looks for in the application binary (using `dlsym`). Additionally, the application can explicitly register the functions using the registration function (also shown in Figure 5.6). Appendix C describes the interfaces to the self CRS in more detail and provides a code example.

### 2. Checkpoint/Restart-Enabled Debugging

The most time consuming part of the software development life-cycle is application debugging. Long-running, large-scale HPC parallel applications compound the time complexity of the debugging process by adding more processes interacting in dynamic ways

```
// Default Checkpoint Callback
int opal_crs_self_user_checkpoint(char **restart_cmd);
// Default Continue Callback
int opal_crs_self_user_continue(void);
// Default Restart Callback
int opal_crs_self_user_restart(void);

// SELF CRS Checkpoint Registration Function
int OMPI_CR_self_register_checkpoint_callback(OMPI_CR_self_checkpoint_fn function);
// SELF CRS Callback Function Signature
typedef int (*OMPI_CR_self_checkpoint_fn)(char **restart_cmd);

// SELF CRS Continue Registration Function
int OMPI_CR_self_register_continue_callback(OMPI_CR_self_continue_fn function);
// SELF CRS Callback Function Signature
typedef int (*OMPI_CR_self_continue_fn)(void);

// SELF CRS Restart Registration Function
int OMPI_CR_self_register_restart_callback(OMPI_CR_self_restart_fn function);
// SELF CRS Callback Function Signature
typedef int (*OMPI_CR_self_restart_fn)(void);
```

FIGURE 5.6. Open MPI self CRS default callbacks, and registration functions.

for longer periods of time. Cyclic or iterative debugging, a commonly used debugging technique, involves repeated program executions that assist the developer in gaining an understanding of the causes of the bug. Software developers can save hours, even days, spent debugging by checkpointing and restarting the parallel debugging session at intermediate points in the debugging cycle. For MPI applications, the parallel debugger must cooperate with the MPI implementation and CRS which account for the network state and process image. In this section we present a design specification for this cooperative relationship to provide C/R-enabled parallel debugging [**133**].

The C/R-enabled parallel debugging design supports multi-threaded MPI applications without requiring any application modifications. Additionally, all checkpoints, whether generated with or without a debugger attached, are both usable within a debugging session as

well as during normal execution. We highlight the *debugger detach* and *debugger re-attach* problems that may lead to inconsistent views of the debugging session, and describe how our design addresses these problems. A discussion of related works is presented in Chapter 2.

**2.1. Design.** C/R-enabled parallel debugging of MPI applications requires the cooperation of the parallel debugger, the MPI implementation, and the CRS to provide consistently recoverable application states. The debugger provides the interface to the user and maintains the state of the parallel debugging session (e.g., breakpoints, watchpoints). Additionally, the debugger may provide the user with additional interfaces to take or return to a checkpoint.

The C/R-enabled MPI implementation marshals the network channels around C/R operations for the application. Though the network channels are often marshaled in a fully coordinated manner, this design does not require full coordination. Therefore the design is applicable to other checkpoint coordination protocol implementations (e.g., uncoordinated).

The CRS captures the state of a single process in the parallel application. This can be implemented at the user- or system-level. This protocol requires an MPI application transparent CRS, which excludes application-level CRSs. If the CRS is not transparent to the application, then taking the checkpoint would alter the state of the program being debugged, potentially confusing the user.

One goal of this design is to create *always usable checkpoints*. This means that regardless of whether the checkpoint was generated with the debugger attached or not, it must be able to be used on restart with or without the debugger. To facilitate the *always usable checkpoints* condition, the checkpoints generated by the CRS with the debugger attached must be able to exclude the debugger state. To achieve this, the debugger must detach from the process before a checkpoint and re-attach, if desired, after the checkpoint has finished, similar to the technique used in [**150**]. Since we are separating the CRS from the debugger, we must consider the needs of both in our design.

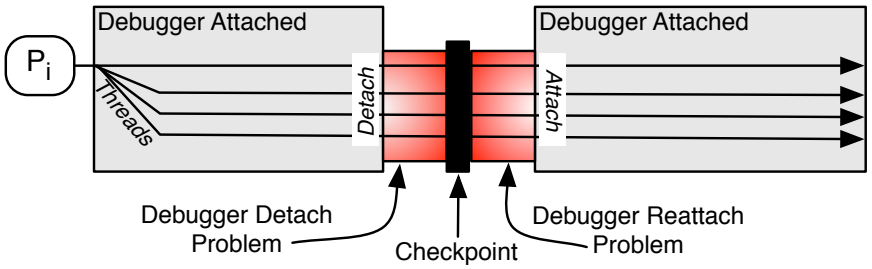



FIGURE 5.7. Illustration of the Debugger Detach and Re-attach Problems.

Detaching the debugger before a checkpoint can allow the application to run uninhibited for a period of time before the actual checkpoint is taken; we call this the *debugger detach problem*. Similarly, once restarted, the MPI application may run uninhibited for a period of time before the debugger attaches; we call this the *debugger re-attach problem*. See Figure 5.7 for an illustration of these problems. Preserving the exact user-perceived state of the program counter in the application across a checkpoint is critical to providing the end user with a seamless and consistent view of the debugging process. Interfaces are prefixed with `MPIR_` to fit the existing naming convention for debugging symbols in MPI implementations.

2.1.1. *Preparing for a Checkpoint.* The C/R-enabled MPI implementation may receive a checkpoint request internally or externally from the debugger, user, or system administrator. The MPI implementation communicates the checkpoint request to the specified processes (usually all processes) in the MPI application. The MPI processes typically prepare for the checkpoint by marshaling the network state and flushing caches before requesting a checkpoint from the CRS.

If the MPI process is under debugger control at the time of the checkpoint, then the debugger must allow the MPI process to prepare for the checkpoint uninhibited by the debugger. If the debugger remains attached, it may interfere with the techniques used by the CRS to preserve the application state (e.g., by masking signals). Additionally, by detaching the debugger before the checkpoint, the implementation can provide the *always usable checkpoints* condition by ensuring that it does not inadvertently include any debugger state in the CRS generated checkpoint.

```
volatile int MPIR_checkpoint_debug_gate = 0;
volatile int MPIR_debug_with_checkpoint = 0;
// Detach Function
int MPIR_checkpoint_debugger_detach(void) {
  return 0;
}
// Thread Wait Function
void MPIR_checkpoint_debugger_waitpoint(void) {
  // MPI Designated Threads are released early,
  // All other threads enter the breakpoint below
  MPIR_checkpoint_debug_gate = 0;
  MPIR_checkpoint_debugger_breakpoint();
}
// Debugger Breakpoint Function
void MPIR_checkpoint_debugger_breakpoint(void) {
  while( MPIR_checkpoint_debug_gate == 0 ) {
    sleep(1);
  }
}
// CRS Hook Callback Function
void MPIR_checkpoint_debugger_crs_hook(int state) {
  if( MPIR_debug_with_checkpoint ) {
    MPIR_checkpoint_debug_gate = 0;
    MPIR_checkpoint_debugger_waitpoint();
  } else {
    MPIR_checkpoint_debug_gate = 1;
  }
}
```

FIGURE 5.8. Debugger `MPIR_` function pseudo code.

The MPI process must inform the debugger of when to detach since the debugger is required to do so before a checkpoint is requested. The MPI process informs the debugger by calling the `MPIR_checkpoint_debugger_detach()` function when it requires the debugger to detach. This is an empty function that the debugger can reference in a breakpoint. It is left to the discretion of the MPI implementation when to call this function while preparing for the checkpoint, but it must be invoked before the checkpoint is requested from the CRS.

```
// CRS hook callback function pseudo code
void MPIR_checkpoint_debugger_crs_hook(int state) {
  if( MPIR_debug_with_checkpoint ) {
    MPIR_checkpoint_debug_gate = 0;
    return MPIR_checkpoint_debugger_waitpoint();
  } else {
    MPIR_checkpoint_debug_gate = 1;
    return NULL;
  }
}
```

FIGURE 5.9. MPI registered CRS hook callback function pseudo code.

The period of time between when the debugger detaches from the MPI process and when the checkpoint is created by the CRS may allow the application to run uninhibited, this is the *debugger detach problem*. See Figure 5.7 for an illustration of the debugger detach problem. To provide a seamless and consistent view to the user, the debugger must make a best effort attempt at preserving the exact position of the program counter(s) across a checkpoint operation. To address the *debugger detach problem*, the debugger forces all threads into a waiting function (called `MPIR_checkpoint_debugger_waitpoint()`) at the current debugging position before detaching from the MPI process. By forcing all threads into a waiting function the debugger prevents the program from making any progress when it returns from the checkpoint operation. Section 2.3.2 describes techniques on how the debugger might achieve this.

The waiting function must allow certain designated threads to complete the checkpoint operation. In a single threaded application, this would be the main thread, but in a multi-threaded application this would be the thread(s) designated by the MPI implementation to prepare for and request the checkpoint from the CRS. The MPI implementation must provide an "early release" check for the designated thread(s) in the waiting function. All other threads directly enter the `MPIR_checkpoint_debugger_breakpoint()` function which waits in a loop for release by the debugger. Designated thread(s) are allowed to continue normal operation, but must enter the breakpoint function after the checkpoint has completed to

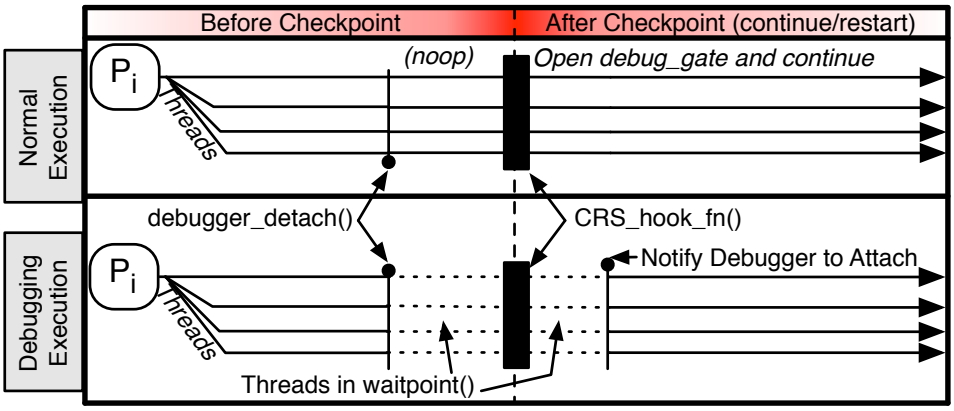



FIGURE 5.10. Illustration of the design for each of the use case scenarios.

provide a consistent state across all threads to the debugger, if it intends on re-attaching. Figure 5.8 presents a pseudo code implementation of these functions.

The breakpoint function loop is controlled by the `MPIR_checkpoint_debug_gate` variable. When this variable is set to `0` the gate is *closed*, keeping threads waiting for the gate to be *opened* by the debugger. To *open* the gate, the debugger sets the variable to a non-zero value, and steps each thread out of the loop and the breakpoint function. Once all threads pass through the gate, the debugger then *closes* it once again by setting the variable back to `0`.

2.1.2. *Resuming After a Checkpoint.* An MPI program either *continues* after a requested checkpoint in the same program, or is *restarted* from a previously established checkpoint saved on stable storage. In both cases the MPI designated thread(s) are responsible for recovering the internal MPI state including reconnecting processes in the network [**135**].

If the debugger intends to attach, the designated thread(s) must inform the debugger when it is safe to attach after restoring the MPI state of the process. If the debugger attaches too early, it may compromise the state of the checkpoint or the restoration of the MPI state. The designated thread(s) notify the debugger that it is safe to attach to the process by printing to `stderr` the hostname and PID of each recovered process. The message is prefixed by "`MPIR_debug_info)`" to distinguish it from other output. Afterwards, the designated thread(s) enter the breakpoint function.

The period of time between when the MPI process is restarted by the CRS and when the debugger attaches may allow the application to run uninhibited, this is the *debugger re-attach problem*. See Figure 5.7 for an illustration of the debugger re-attach problem. If the MPI process was being debugged before the checkpoint was requested, then the threads are already being held in the breakpoint function, thus preventing them from running uninhibited. However, if the MPI process was not being debugged before the checkpoint then the user may experience inconsistent behavior due to the race to attach the debugger upon multiple restarts from the same checkpoint.

To address this problem, the CRS must provide a *hook* callback function that is pushed onto the stack of all threads before returning them to the running state. This technique preserves the individual thread's program counter position at the point of the checkpoint providing a best effort attempt at a consistent recovery position upon multiple restarts. The MPI implementation registers a hook callback function that will place all threads into the waiting function if the debugger intends to re-attach. The intention of the debugger to re-attach is indicated by the `MPIR_debug_with_checkpoint` variable. Since the hook function is the same function used when preparing for a checkpoint, the release of the threads from the waiting function is consistent from the perspective of the debugger.

If the debugger is not going to attach after the checkpoint or on restart, the hook callback does not need to enter the waiting function, again indicated by the `MPIR_debug_with_checkpoint` variable. Since the checkpoint could have been generated with a debugger previously attached, the hook function must release all threads from the breakpoint function by setting the `MPIR_checkpoint_debug_gate` variable to `1`. The structure of the hook callback function allows for checkpoints generated while debugging to be used without debugging, and vice versa. See Figure 5.8 for an example of the hook callback function. The MPI implementation may want to prevent the threads from entering the waiting function multiple times by determining if the threads were already placed in the waiting function during checkpoint preparation.

2.1.3. *Additional* `MPIR_` *Symbols.* In addition to the detach, waiting, and breakpoint functions, this design defines a series of support variables to allow the debugger greater

```
(gdb) p MPIR_checkpointable
$1 = 1
(gdb) p MPIR_checkpoint_command
$1 = ompi-checkpoint --crdebug --hnp-jobid 1234
(gdb) p MPIR_restart_command
$1 = ompi-restart crdebug
(gdb) p MPIR_checkpoint_listing_command
$1 = ompi-checkpoint -l --crdebug
(gdb) p MPIR_controller_hostname
$1 = localhost
(gdb) set MPIR_debug_with_checkpoint = 0
(gdb) detach
```

FIGURE 5.11. Additional `MPIR_` symbols.

generality when interfacing with a C/R-enabled MPI implementation. These interfaces, with their Open MPI representation, are presented in Figure 5.11 and described in this section.

The `MPIR_checkpointable` variable indicates to the debugger that the MPI implementation is C/R-enabled and supports this design when set to `1`. The `MPIR_debug_with_checkpoint` variable indicates to the MPI implementation if the debugger intends to attach. If the debugger wishes to detach from the program, it sets this value to `0` before detaching indicating that it no longer intends to debug the program. This value is set to `1` when the debugger attaches to the job either while running or on restart.

The `MPIR_checkpoint_command` variable specifies the command to be used to initiate a checkpoint of the MPI process. The output of the checkpoint command must be formatted such that the debugger can use it directly as an argument to the restart command. The output on `stderr` is prefixed with "`MPIR_checkpoint_handle)`" as to distinguish it from other output. The `MPIR_restart_command` variable specifies the restart command to prefix the output of the checkpoint command to restart an MPI application. The `MPIR_controller_hostname` variable specifies the host on which to execute the `MPIR_checkpoint_command` and `MPIR_restart_command` commands. The `MPIR_checkpoint_listing_command` variable specifies the command that lists the available checkpoints on the system.

| | Scenario 1 | | Scenario 2 | | Scenario 3 | | Scenario 4 | |
|---|---|---|---|---|---|---|---|---|
| Before/After the Checkpoint | Pre | Post | Pre | Post | Pre | Post | Pre | Post |
| `MPIR_checkpointable` | T | T | T | T | T | T | T | T |
| `MPIR_debug_with_checkpoint` | F | F | F | T | T | T | T | F |
| `MPIR_checkpoint_debug_gate` | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| Threads waiting? | F | F | F | T | T | T | T | F |

TABLE 5.2. `MPIR_` variable states associated with use case scenarios.

**2.2. Use Case Scenarios.** To better illustrate how the various components cooperate to provide C/R-enabled parallel debugging we present a set of use case scenarios. Table 5.2 describes the state of the various `MPIR_` variables both before a checkpoint occurs and afterwards (either in a continue or restart states). Figure 5.10 presents an illustration of the design for each scenario.

2.2.1. *Scenario 1: No Debugger Involvement.* This is the standard C/R scenario in which the debugger is neither involved before a checkpoint nor afterwards (a transition from the upper-left to upper-right quadrants in Figure 5.10). The MPI processes involved in the checkpoint will prepare the internal MPI state and request a checkpoint from the CRS then continue free execution afterwards. Before requesting a checkpoint the MPI designated checkpoint thread(s) call the `MPIR_checkpoint_debugger_detach()` function, which has no affect on the process since a debugger is not attached.

After the checkpoint is finished, the hook callback function is executed in each thread in the MPI process by the CRS. Since no debugger intends on attaching, these functions exit without waiting and the MPI program is allowed to continue execution as normal.

2.2.2. *Scenario 2: Debugger Attaches on Restart.* In this scenario, the debugger is attaching to a restarting MPI process from a checkpoint that was generated without the debugger (a transition from the upper-left to lower-right quadrants in Figure 5.10). This scenario is useful when repurposing checkpoints originally generated for fault tolerance purposes instead for debugging. The process of creating the checkpoints is the same as in Scenario 1. On restart, the hook callback function is called by the CRS in each thread to preserve their program counter positions.

When the MPI process is restarted, the hook callback function is executed in each thread in the MPI process by the CRS, thus preserving the position of the program counter in each thread consistently at the point of the checkpoint. The MPI designated checkpoint thread(s) are allowed to exit the hook function to reconstruct the MPI state, while all other threads wait in the breakpoint function. After reconstructing the MPI state the designated thread(s) indicate to the debugger that it is safe to attach (on `stderr`), and enter the breakpoint function.

The debugger then attaches to the MPI process and walks all threads out of the breakpoint function. It is then in full control of the MPI process and able to resume debugging at the same point in the application execution for every restart from the same checkpoint.

2.2.3. *Scenario 3: Debugger Attached While Checkpointing.* In this scenario, the debugger is attached when a checkpoint is requested of the MPI process (a transition from the lower-left to lower-right quadrants in Figure 5.10). This scenario is useful when creating checkpoints while debugging that can be returned to in later iterations of debugging cycle or to provide backstepping functionality while debugging. The debugger will notice the call to the detach function and call the waiting function in all threads in the MPI process.

The MPI processes involved in the checkpoint will prepare for the internal MPI state and the MPI designated thread(s) call the detach function just before requesting the checkpoint from the CRS. Alternatively, the detach function may be called before the MPI designated thread(s) prepare the internal MPI state, whenever makes the most sense for the MPI implementation. The debugger will notice the call to the detach function and call the waiting function in all threads in the MPI process. The MPI designated checkpoint thread(s) are allowed to continue through this function in order to request the checkpoint from the CRS while all other threads wait there for later release.

After the checkpoint is finished, the hook callback function is executed in each thread in the MPI process by the CRS. Since all threads were placed in the breakpoint function before the checkpoint was requested there is no need to reenter the function, so threads may be allowed to fall though this function and continue waiting at the previous breakpoint function call. After reconstructing the MPI state, the designated checkpoint thread(s) indicate to the

debugger that it is safe to reattach (via message on `stderr`, see Section 2.1.2) then enter the breakpoint function. The debugger then attaches to the MPI process as in Scenario 1.

2.2.4. *Scenario 4: Debugger Detached on Restart.* In this scenario, the debugger is attached when the checkpoint is requested of the MPI process, but is not when the MPI process is restarted from the checkpoint (a transition from the lower-left to upper-right quadrants in Figure 5.10). This scenario is useful when analyzing the uninhibited behavior of an application, periodically inspecting checkpoints for validation purposes or, possibly, introducing tracing functionality to a running program. The process of creating the checkpoint is the same as in Scenario 3. By inspecting the `MPIR_debug_with_checkpoint` variable, the MPI processes know to let themselves out of the waiting function afterwards.

When the MPI process is restarted, the hook callback function is executed in each thread in the MPI process by the CRS. Since the debugger does not intend to attach, the threads are allowed to exit the function without waiting. In order to escape the breakpoint function that the threads were placed in by the debugger before the checkpoint, they must open the `MPIR_checkpoint_debug_gate` before exiting the hook function. The debugger will reset this value when, if ever, it intends on attaching at a later point in time.

**2.3. Implementation.** The design described in this section was implemented using GNU's GDB debugger, Allinea's DDT Parallel Debugger, the Open MPI implementation of the MPI standard, and the BLCR Checkpoint/Restart Service (CRS). Open MPI implements a fully coordinated C/R protocol [**136**] so when a checkpoint is requested of one process all processes in the MPI job are also checkpointed. It should be pointed out again that nothing about the specification requires a coordinated protocol, so other coordination protocols (e.g., uncoordinated) can be used at the discretion of the MPI implementation.

2.3.1. *Interlayer Notification Callback Functions.* The Interlayer Notification Callback (INC) functions are used in Open MPI to coordinate the internal state of the MPI implementation before and after checkpoint operation. After receiving notification of a checkpoint request, Open MPI calls the `INC_checkpoint_prep()` function (See Figure 5.12). This function quiesces the network, and prepares various components for a checkpoint

```
void INC_checkpoint_prep() {
  // Prepare the network and MPI state
  Quiesce_Network();
  Prep_Components();
  // Identify this thread as a special thread
  free_threads[0] = thread_self();
  // Detach the Debugger
  MPIR_checkpoint_debugger_detach();
  // Request the checkpoint
  CRS_request_checkpoint();
}
```

FIGURE 5.12. Pseudo code of Open MPI's checkpoint preparation INC function.

operation [**135**]. Once the INC is finished, it designates a checkpoint thread, calls the `MPIR_checkpoint_debugger_detach()` function, and requests the checkpoint from the CRS, in this case BLCR.

After the checkpoint is created (called the *continue* state), or when the MPI process is restarted (called the *restart* state), BLCR calls the hook callback function in each thread (See Figure 5.8). The thread designated by Open MPI in the `INC_checkpoint_prep()` function is allowed to exit this function without waiting, while all other threads must wait if the debugger intends on attaching. The designated thread then calls the INC function for either the continue or the restart phase, depending on if the MPI process is continuing after a checkpoint or restarting from a checkpoint previously saved to stable storage. In Open MPI, these functions operate in a similar manner, though differ slightly in complexity of operation. In the continue phase, the MPI implementation can assume that most of the cached system and network information remains valid, while in the restart phase it cannot make such assumptions since processes have likely moved around in the system. For brevity, we present the pseudo code for the INC function that is applicable for both of these states in Figure 5.13.

If the debugger intends on attaching to the MPI process, then after reconstructing the MPI state, the designated thread notifies the debugger that it is safe to attach by printing to

```
void INC_checkpoint_recovery() {
  // Reattach processes
  Recover_Network();
  Recover_Components();
  // If the debugger is going to reattach
  if( MPIR_debug_with_checkpoint ) {
    // Notify Debugger that it is ok to reattach
    // MPIR_debug_info) localhost:789
    print_reattach_message();
    // Wait for Debugger to reattach
    MPIR_checkpoint_debugger_breakpoint();
  }
}
```

FIGURE 5.13. Pseudo code of Open MPI's checkpoint continue/restart INC function.

`stderr` the hostname and PID of each recovered process prefixed with "`MPIR_debug_info)`" as mentioned in Section 2.1.2. The debugger can then attach and walk the threads out of the breakpoint function (by *opening* the debug gate) and resume debugging. If the debugger does not intend to attach to the MPI process, then after reconstructing the MPI state the designated thread releases all the other threads and resumes normal execution.

2.3.2. *Stack Modification.* In Section 2.1.1, the debugger was required to force all threads to call the waiting function before detaching before a checkpoint in order to preserve the program counter in all threads across a checkpoint operation. We explored two different ways to do this in the GDB debugger. The first required the debugger to force the function on the call stack of each thread. In GDB we executed the the following command in each thread:

```
call MPIR_checkpoint_debugger_waitpoint()
```

Unfortunately this became brittle and corrupted the stack in GDB 6.8.

In response to this, we explored an alternative technique based on signals. Open MPI registered a signal callback function (See Figure 5.14) that calls the `MPIR_checkpoint_debugger_waitpoint()` function. The debugger can then send a designated signal (e.g.,

```
// Signal handler for SIGTSTP
void MPIR_checkpoint_debugger_signal_handler(int num) {
  MPIR_checkpoint_debugger_waitpoint();
}
```

FIGURE 5.14. Open MPI's signal handler function to support stack modification.

SIGTSTP) to each thread in the application, and the program will place itself in the waiting function.

Though the signal based technique worked best for GDB, other debuggers may have other techniques at their disposal to achieve this goal.

## 3. Conclusion

This chapter focuses on presenting interfaces to the C/R functionality (described in Chapter 3 and 4) to applications and parallel debuggers. The application interfaces allow an application to define quiescent regions, request checkpoints, restarts and process migrations. The INC callbacks allow an application to participate in the checkpoint and restart life-cycle. The self CRS even allows the application layer C/R mechanisms to take advantage of the coordination (via SnapC) and storage (via SStore) features provided by Open MPI.

The C/R-enabled parallel debugging protocol reduces the time spent debugging long-running applications by returning a developer to an intermediary point in the computation closer to the bug. The discussion of this protocol highlighted the debugger detach and reattach problems and presented solution that preserver the always usable checkpoints goal.

# 6

# Conclusions

Our research defined seven capabilities that, when taken together, form a complete transparent, coordinated C/R fault tolerance infrastructure for MPI implementations to support resilient MPI applications on HPC systems. Our implementation of these seven capabilities in Open MPI provided end users with three often requested C/R-related services in a single implementation. First, we provided applications with automatic and manual reactive fault recovery services, allowing them to recover from unexpected process failure. Next, we provided applications with a proactive process migration service that allows them to move processes away from anticipated failure. Lastly, we provided a protocol for C/R-enabled parallel debugging that reduces the time spent debugging long-running applications by allowing the developer to return to an intermediary point in the computation closer to the

bug. A significant contribution of our transparent C/R infrastructure is its ability to turn a traditional MPI application into a resilient MPI application without application modification. Our implementation in Open MPI is unique in that it provides reactive fault recovery, proactive process migration, and C/R-enabled parallel debugging services within a single implementation with clearly defined capabilities.

Application developers are concerned about the performance impact of incorporating any fault tolerance techniques into their application, particularly transparent fault tolerance techniques. With this in mind, we investigated the performance implications of our implementation on a variety of benchmarks and real applications including HPL, POP, LAMMPS, NAS, and GROMACS. We verified that stable storage is the predominate C/R performance bottleneck, representing up to 98% of the checkpoint overhead in a traditional configuration. We demonstrated a significant reduction in checkpoint overhead (up to 38% for some applications) by replacing the traditional stable storage approach with a staging model that overlapped checkpoint establishment with normal computation. We assessed the impact of using node-local caching and compression on automatic recovery and checkpoint overhead. We found that node-local caching can significantly improve the performance and scalability of automatic recovery. We showed that including off-line, node-local compression in the staging pipeline not only saved stable storage space, but also improved the checkpoint latency and overhead (up to 72% and 40%, respectively, for some applications). Post-restart application performance is just as important as checkpoint performance. With this in mind, we demonstrated a technique that preserves performance across process recovery by allowing rediscovery of interconnects between processes to better adapt to availability and process layout adjustments.

Applications can often further improve the performance of a C/R implementation by guiding the checkpoint, restart and migration operations. Our implementation in Open MPI provides a set of APIs and command line tools that were designed to increase end user adoption and third party software integration. Such interfaces allow the application to, optionally, break the transparency barrier to customize the C/R solution to the application's reliability and performance requirements.

The C/R infrastructure in Open MPI separates each of the seven capabilities into individual frameworks; this makes the entire infrastructure more maintainable for MPI developers, and extensible for fault tolerance researchers. Future implementers can build upon our research to create more maintainable, extensible, and flexible designs. Further, our modular design lowers the bar to entry for C/R researchers by allowing them to focus their development on a subset of capabilities, spurring innovation. Lastly, applications can immediately take advantage of this ongoing research by using our C/R-enabled MPI infrastructure in Open MPI.

The capabilities defined for C/R fault tolerance are proving useful in supporting research into alternative fault tolerance techniques. The run-through stabilization ErrMgr component is being used to support research into fault tolerant MPI semantics. The non-blocking process management MPI interface presented in this dissertation, combined with the stabilization ErrMgr component, will allow an application to overlap process recovery with normal computation, preserving performance during recovery. As HPC reliability declines for long-running and scalable scientific applications, fault tolerance researchers must quickly adapt to an application's reliability requirements. A composable design with clearly identifiable capabilities allows fault tolerance researchers to match their research efforts to the pace of the application's reliability requirements.

## 1. Future Work

Future extensions to the work presented in this dissertation is inevitable as technology advances and research scrambles to adapt to future HPC systems. As applications begin to experiment with MPI-2 and future MPI standard interfaces, implementations of the capabilities presented in this dissertation will likely need to be extended and the relationship between capabilities will need to be reassessed. For example, one-sided communication and parallel I/O operations will require special attention when checkpointing to preserve performance and account for file contents.

CRS implementations are becoming more available and maturing to include advanced features such as memory inclusion/exclusion and incremental checkpointing [**209**]. The

CRS implementation in Open MPI will need to evolve to incorporate these features and expose them to the application as necessary. The interaction between the CRS and SStore capabilities will need to be adapted to better support incremental checkpointing since a checkpoint carries a dependency on previous incremental checkpoints.

Since stable storage is the primary bottleneck in a C/R solution, future work should investigate advanced staging and peer-based storage techniques. This dissertation highlighted some of the benefits of including off-line compression in the staging pipeline. A formal model of the impact of compression in a staging pipeline will assist end users in determining if and how compression might benefit their application. Future analysis of compression techniques may wish to consider the benefits of using intermediary nodes and alternative compression techniques. Future work into checkpoint storage should investigate the benefits of peer-based storage (e.g., SCR [**36**]) and checkpoint-specific file systems (e.g., stdchk [**5**]). Future work may also extend the implementation of the FileM capability by supporting standard UNIX copy commands and high performance out-of-band communication channels for checkpoint file and directory management.

Even though stable storage is the primary bottleneck in a C/R solution, future work should also explore alternative checkpoint coordination algorithms in the CRCP capability. Alternative checkpoint coordination algorithms often allow for looser synchrony between processes which may lead to a more scalable C/R solution [**103, 263, 266**].

Due to the checkpoint unfriendly nature of high-speed interconnects like InfiniBand and Myrinet the Open MPI implementation is forced to shut down these interconnects at every checkpoint request. Future work should improve the interaction between the interconnect drivers and CRSs to reduce the checkpoint overhead by eliminating the need to shut down and reconnect these drivers across a checkpoint operation.

In this dissertation we assessed the impact of an *eager* process migration protocol. Future work may wish to investigate direct copy techniques that remove the logically centralized stable storage device from the process migration procedure in our implementation. Future work may also wish to investigate other process migration protocols like *pre-copy* [**257, 277**] and *quasi-asynchronous* [**66**]. These techniques can potentially reduce the

migration overhead by overlapping the migration of a process with the transfer of state and normal computation.

# Bibliography


[1] Mostafa Abd-El-Barr. *Design And Analysis of Reliable And Fault-tolerant Computer Systems*. Imperial College Press, 2007.

[2] Hazim Abdel-Shafi, Evan Speight, and John K. Bennett. Efficient user-level thread migration and checkpointing on Windows NT clusters. In *WINSYM'99: Proceedings of the 3rd conference on USENIX Windows NT Symposium*, Berkeley, CA, USA, 1999. USENIX Association.

[3] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 277–286, New York, NY, USA, 2004. ACM.

[4] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, 8(3):21–26, 1991.

[5] Samer Al-Kiswany, Matei Ripeanu, Sudharshan S. Vazhkudai, and Abdullah Gharaibeh. stdchk: A checkpoint storage system for desktop grid computing. *International Conference on Distributed Computing Systems*, pages 613–624, 2008.

[6] Peter A. Alsberg and John D. Day. A principle for resilient sharing of distributed resources. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[7] Lorenzo Alvisi, B. Hoppe, and Keith Marzullo. Nonblocking and orphan-free message logging protocols. In *The 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 145–154, Toulouse, France, June 1993.

[8] Lorenzo Alvisi and Keith Marzullo. Trade-offs in implementing causal message logging protocols. In *PODC '96: Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 58–67, New York, NY, USA, 1996. ACM.

[9] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *IEEE Transactions on Software Engineering*, 24(2):149–159, 1998.

[10] Lorenzo Alvisi, Sriram Rao, Syed Amir Husain, Asanka de Mel, and Elmootazbellah Elnozahy. An analysis of communication-induced checkpointing. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, Washington, DC, USA, 1999. IEEE Computer Society.

[11] Jason Ansel, Kapil Aryay, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. *International Parallel and Distributed Processing Symposium*, pages 1–12, 2009.

[12] Jason Ansel, Michael Rieker, and Gene Cooperman. User-level socket-based checkpointing for distributed and parallel computation. *The Computing Research Repository*, 0701037, 2007.

[13] Sarala Arunagiri, John T. Daly, and Patricia J. Teller. Modeling and analysis of checkpoint I/O operations. In *ASMTA '09: Proceedings of the 16th International Conference on Analytical and Stochastic Modeling Techniques and Applications*, pages 386–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] Algirdas Avizienis. Fault-tolerance and fault-intolerance: Complementary approaches to reliable computing. In *Proceedings of the International Conference on Reliable Software*, pages 458–464, New York, NY, USA, 1975. ACM Press.

[15] Wendy Bartlett and Lisa Spainhower. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable Secure Computing*, 1(1):87–96, 2004.

[16] Rajanikanth Batchu, Anthony Skjellum, Zhenqian Cui, Murali Beddhu, Jothi P. Neelamegam, Yoginder Dandass, and Manoj Apte. MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *CCGRID '01: Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, 2001. IEEE Computer Society.

[17] Adam Beguelin, Erik Seligman, and Peter Stephan. Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing*, 43(2):147–155, 1997.

[18] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A checkpoint filesystem for parallel applications. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[19] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A checkpoint filesystem for parallel applications. Technical Report LA-UR-09-02117, Los Alamos National Laboratory, April 2009.

[20] Marin Bertier, Olivier Marin, and Pierre Sens. Performance analysis of a hierarchical failure detector. *International Conference on Dependable Systems and Networks*, 2003.

[21] K. Bhatia, K. Marzullo, and L. Alvisi. The relative overhead of piggybacking in causal message logging protocols. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, Washington, DC, USA, 1998. IEEE Computer Society.

[22] Karan Bhatia, Keith Marzullo, and Lorenzo Alvisi. Scalable causal message logging for wide-area environments. *Proceedings of the European Conference on Parallel Computing (Euro-Par 2001)*, pages 864–873, 2001.

[23] Bob Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 299–310, New York, NY, USA, 2000. ACM Press.

[24] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton 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.

[25] Fatiha Bouabache, Thomas Herault, Gilles Fedak, and Franck Cappello. Hierarchical replication techniques to ensure checkpoint storage reliability in grid environment. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 475–483, Washington, DC, USA, 2008. IEEE Computer Society.

[26] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Retrospect: Deterministic replay of MPI applications for interactive distributed debugging. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 297–306, 2007.

[27] Aurelien Bouteiller, Franck Cappello, Thomas Herault, Geraud Krawezik, Pierre Lemarinier, and Frederic Magniette. MPICH-V2: A fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2003. IEEE Computer Society.

[28] Aurelien Bouteiller, Boris Collin, Thomas Herault, Pierre Lemarinier, and Franck Cappello. Impact of event logger on causal message logging protocols for fault tolerant MPI. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Washington, DC, USA, 2005. IEEE Computer Society.

[29] Aurelien Bouteiller, Thomas Herault, G. Krawezik, Pierre Lemarinier, and Franck Cappello. MPICH-V project: A multiprotocol automatic fault-tolerant MPI. In *International Journal of High Performance Computing Applications*, volume 20, pages 319–333. Sage Publications, Inc., 2006.

[30] Aurelien Bouteiller, Pierre Lemarinier, Geraud Krawezik, and Franck Cappello. Coordinated checkpoint versus message log for fault tolerant MPI. *IEEE International Conference on Cluster Computing*, 2003.

[31] M. Bozyigit and M. Wasiq. User-level process checkpoint and restore for migration. *SIGOPS Operating Systems Review*, 35(2):86–96, 2001.

[32] Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Recent advances in checkpoint/recovery systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2006. Next Generation Systems Program Workshop.

[33] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *PPoPP '03: Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 84–94, New York, NY, USA, 2003. ACM Press.

[34] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. $C^3$: A system for automating application-level checkpointing of MPI programs. *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computers (LCPC'03)*, pages 357–373, 2003.

[35] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Collective operations in application-level fault-tolerant MPI. In *ICS '03: Proceedings of the 17th Annual International Conference on Supercomputing*, pages 234–243, New York, NY, USA, 2003. ACM Press.

[36] Greg Bronevetsky and Adam Moody. Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O. Technical Report LLNL-TR-415791, Lawrence Livermore National Laboratory, 2009.

[37] Clairton Buligon, Sergio Cechin, and Ingrid Jansch-Porto. Implementing rollback-recovery coordinated checkpoints. *Advanced Distributed Systems*, 3563:246–257, August 2005.

[38] Darius Buntinas, Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI protocols. *Future Generation Computer Systems*, 24(1), January 2008.

[39] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[40] A. Calderon, F. Garcia-Carballeira, J. Carretero, J. M. Perez, and L. M. Sanchez. A fault tolerant MPI-IO implementation using the expand parallel file system. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 274–281, Washington, DC, USA, 2005. IEEE Computer Society.

[41] Franck Cappello, Al Geist, Bill Gropp, Laxmikant Kale, Bill Kramer, and Marc Snir. Toward exascale resilience. *International Journal of High Performance Computing Applications*, 23(4):374–388, 2009.

[42] Jeremy Casas, Dan Clark, Phil Galbiati, Ravi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole. MIST: PVM with transparent migration and checkpointing. In *In 3rd Annual PVM Users' Group Meeting*, 1995.

[43] Jeremy Casas, Dan Clark, Rabi Konuru, Steve Otto, Robert Prouty, and Jonathan Walpole. MPVM: A migration transparent version of PVM. Technical Report CSE-95-002, Oregon Graduate Institute School of Science and Engineering, February 1995.

[44] Miguel Castro, Rodrigo Rodrigues, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions Computer Systems*, 21(3):236–269, 2003.

[45] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. *SIGPLAN Notices*, 26(4):200–211, 1991.

[46] Sayantan Chakravorty and L. V. Kale. A fault tolerance protocol with fast fault recovery. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium*. IEEE Press, 2007.

[47] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[48] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions Computer Systems*, 3(1):63–75, 1985.

[49] Yun Seok Chang, Sun Young Cho, and Bo Yeon Kim. Performance evaluation of the striped checkpointing algorithm on the distributed RAID for cluster computer. In *Computational Science (ICCS 2003)*, volume 2658, pages 955–962, January 2003.

[50] Kulathep Charoenpornwattana, Chokchai Leangsuksun, Geoffroy Vallée, Anand Tikotekar, and Stephen L. Scott. A neural networks approach for intelligent fault prediction in HPC environments. In *HAPCW'08: High Availability and Performance Computing Workshop*, Denver, Colorado, USA, 2008.

[51] Yuqun Chen, James S. Plank, and Kai Li. CLIP: A checkpointing tool for message-passing parallel programs. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–11, New York, NY, USA, 1997. ACM.

[52] Zizhong Chen and Jack Dongarra. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems*, 19(12):1628–1641, 2008.

[53] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Building fault survivable MPI programs with FT-MPI using diskless checkpointing. Technical Report UT-CS-04-540, University of Tennessee, December 2004.

[54] Zizhong Chen, Graham E. Fagg, Edgar Gabriel, Julien Langou, Thara Angskun, George Bosilca, and Jack Dongarra. Fault tolerant high performance computing by a coding approach. In *PPoPP '05: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[55] Zizhong Chen, Ming Yang, Guillermo Francia, and Jack Dongarra. Self adaptive application level fault tolerance for parallel and distributed computing. *IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[56] Kwang-Sik Chung, Kibom Kim, Chong-Sun Hwang, Jin Gon Shon, and HeonChang Yu. Hybrid checkpointing protocol based on selective-sender-based message logging. In *ICPADS '97: Proceedings of the*

*1997 International Conference on Parallel and Distributed Systems*, pages 788–793, Washington, DC, USA, 1997. IEEE Computer Society.

[57] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006. ACM.

[58] James Cownie and William Gropp. A standard interface for debugger access to message queue information in MPI. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 51–58, London, UK, 1999. Springer-Verlag.

[59] Antonio Cunei and Jan Vitek. A new approach to real-time checkpointing. In *VEE '06: Proceedings of the 2nd International Conference on Virtual Execution Environments*, pages 68–77, New York, NY, USA, 2006. ACM.

[60] Bill Curtis. Fifteen years of psychology in software engineering: Individual differences and cognitive science. In *ICSE '84: Proceedings of the 7th international conference on Software engineering*, pages 97–106, Piscataway, NJ, USA, 1984. IEEE Press.

[61] Paweł Czarnul and Marcin Fraczak. New user-guided and ckpt-based checkpointing libraries for parallel MPI applications. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 3666:351–358, September 2005.

[62] Pawel Czarnul, Arkadiusz Urbaniak, Marcin Fraczak, Maciej Dyczkowski, and Bartlomiej Balcerek. Towards easy-to-use checkpointing of MPI applications within CLUSTERIX. In *PARELEC '04: Proceedings of the International Conference on Parallel Computing in Electrical Engineering*, pages 390–393, Washington, DC, USA, 2004. IEEE Computer Society.

[63] John T. Daly. A strategy for running large scale applications based on a model that optimizes the checkpoint interval for restart dumps. *Software Engineering for High Performance Computing System (HPCS) Applications at the 26th International Conference on Software Engineering*, pages 70–74, May 2004.

[64] John T. Daly. Quantifying checkpoint efficiency. Technical Report LALP-07-041, Los Alamos National Laboratory, 2007.

[65] Suresh K. Damodaran-Kamal and Joan M. Francioni. Nondeterminancy: Testing and debugging in message passing parallel programs. In *PADD '93: Proceedings of the 1993 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 118–128, New York, NY, USA, 1993. ACM.

[66] Pei Dan, Wang Dongsheng, Zhang Youhui, and Shen Meiming. Quasi-asynchronous migration: A novel migration protocol for PVM tasks. *SIGOPS Operating Systems Review*, 33(2):5–14, 1999.

[67] Raphael Y. de Camargo, Fabio Kon, and Renato Cerqueira. Strategies for checkpoint storage on opportunistic grids. *IEEE Distributed Systems Online*, 7(9), 2006.

[68] Jacques Chassin de Kergommeaux, Michiel Ronsse, and Koenraad De Bosschere. MPL*: Efficient record/-play of nondeterministic features of message passing libraries. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 141–148, London, UK, 1999. Springer-Verlag.

[69] Alan Demers, Dan Greene, Carl Houser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *SIGOPS Operating Systems Review*, 22(1):8–32, 1988.

[70] William R. Dieter and James E. Lumpp, Jr. User-level checkpointing for LinuxThreads programs. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 81–92, Berkeley, CA, USA, 2001. USENIX Association.

[71] William R. Dieter and James E. Lumpp Jr. A user-level checkpointing library for POSIX threads programs. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, Washington, DC, USA, 1999. IEEE Computer Society.

[72] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[73] Xiangyu Dong, Naveen Muralimanohar, Norm Jouppi, Richard Kaufmann, and Yuan Xie. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[74] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. A case study of incremental and background hybrid in-memory checkpointing. *The Exascale Evaluation and Research Techniques Workshop in conjunction with 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[75] Yunfei Du, Panfeng Wang, Hongyi Fu, Jia Jia, Haifang Zhou, and Xuejun Yang. Building single fault survivable parallel algorithms for matrix operations using redundant parallel computation. *International Conference on Computer and Information Technology*, pages 285–290, 2007.

[76] Angelo Duarte, Delores Rexachs, and Emilio Luque. An intelligent management of fault tolerance in cluster using RADICMPI. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 4192:150–157, 2006.

[77] Andrzej Duda. The effects of checkpointing on program execution time. *Information Processing Letters*, 16:221–229, June 1983.

[78] 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, December 2002.

[79] Jason Duell, Paul Hargrove, and Eric Roman. Requirements for Linux checkpoint/restart. Technical Report LBNL-49659, Lawrence Berkeley National Laboratory, 2002.

[80] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[81] Elmootazbellah N. Elnozahy. How safe is probabilistic checkpointing? In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, Washington, DC, USA, 1998. IEEE Computer Society.

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

[83] Elmootazbellah N. Elnozahy, David B. Johnson, and Willy Zwaenpoel. The performance of consistent checkpointing. In *Proceedings of the 11th IEEE Symposium on Reliable Distributed Systems*, pages 39–47, Houston, Texas, October 1992.

[84] Elmootazbellah N. Elnozahy and James S. Member-Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Transactions on Dependable Secure Computing*, 1(2):97–108, 2004.

[85] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[86] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Replicated distributed processes in Manetho. In *Proceedings of the Twenty Second Annual International Symposium on Fault-Tolerant Computing, FTCS-22*, pages 18–27, July 1992.

[87] Jakob Engblom. Checkpointing. Technical report, VirtuTech, July 2009.

[88] Christian Engelmann and Al Geist. Super-scalable algorithms for computing on 100,000 processors. In *Proceedings of International Conference on Computational Science (ICCS)*, volume 3514, pages 313–320, May 2005.

[89] Christian Engelmann and Stephen L. Scott. High availability for ultra-scale high-end scientific computing. In *Proceedings of the 2nd International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2) 2005, in conjunction with the 19th ACM International Conference on Supercomputing (ICS) 2005*, Cambridge, MA, USA, 2005.

[90] Christian Engelmann, Geoffroy R. Vallee, Thomas Naughton, and Stephen L. Scott. Proactive fault tolerance using preemptive migration. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, pages 252–257, 2009.

[91] M. Rasit Eskicioglu. Design issues of process migration facilities in distributed systems. In *Scheduling and Load Balancing in Parallel and Distributed Systems*, pages 414–424, 1995.

[92] Graham E. Fagg, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J. Dongarra. Scalable fault tolerant MPI: Extending the recovery algorithm. In *Proceedings of Recent Advances in Parallel Virtual Machine and Messaging Passing Interface Users' Group Meeting Euro PVMMPI*, pages 67–75. Springer Heidelberg, Lecture Notes in Computer Science, 2005.

[93] Graham E. Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J. Dongarra. Process fault-tolerance: Semantics, design and applications for high performance computing. *International Journal for High Performance Applications and Supercomputing*, 19(4):465–478, 2005.

[94] Michael Feldman. The other exascale challenge, June 2010. `http://www.hpcwire.com/blogs/The-Other-Exascale-Challenge-96104409.html`.

[95] Stuart I. Feldman and Channing B. Brown. IGOR: A system for program debugging via reversible execution. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, New York, NY, USA, 1988. ACM Press.

[96] Rohit Fernandes, Keshav Pingali, and Paul Stodghill. Mobile MPI programs in computational grids. *ACM 2006 Symposium on Principles and Practice of Parallel Programming*, March 2006.

[97] Kurt Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretti, Todd Kordenbrock, and Ron Brightwell. Increasing fault resiliency in a message-passing environment. Technical Report SAND2009-6753, Sandia National Laboratories, October 2009.

[98] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[99] Mark S. Foster and Joseph N. Wilson. Pursuing the three AP's to checkpointing with UCLiK. In *10th International Linux System Technology Conference*, 2003.

[100] Song Fu and Cheng-Zhong Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[101] Future Technologies Group. Berkeley Lab Checkpoint/Restart (BLCR). `http://ftg.lbl.gov/checkpoint/`.

[102] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[103] Qi Gao, Wei Huang, M. Koop, and Dhabaleswar K Panda. Group-based coordinated checkpointing for MPI: A case study on InfiniBand. *In ICPP'06: Proceedings of the 35th International Conference on Parallel Processing*, January 2007.

[104] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. *International Conference on Parallel Processing*, pages 471–478, Jan 2006.

[105] Wen Gao, Mingyu Chen, and Takashi Nanya. A faster checkpointing and recovery algorithm with a hierarchical storage approach. In *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, Washington, DC, USA, 2005. IEEE Computer Society.

[106] Rahul Garg, Vijay K. Garg, and Yogish Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*, pages 269–277, New York, NY, USA, 2006. ACM.

[107] Al Geist and Christian Engelmann. Development of naturally fault tolerant algortihms for computing on 100,000 processors. *Journal of Parallel and Distributed Computing*, 2002.

[108] Erol Gelenbe. A model of roll-back recovery with multiple checkpoints. In *ICSE '76: Proceedings of the 2nd International Conference on Software Engineering*, pages 251–255, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[109] Erol Gelenbe. On the optimum checkpoint interval. *Journal of the ACM*, 26(2):259–270, 1979.

[110] Stephane Genaud and Choopan Rattanapoka. P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs on grids. *Journal of Grid Computing*, 5(1):27–42, 2007.

[111] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *SIGOPS Operating Systems Review*, 37(5):29–43, 2003.

[112] David K. Gifford. Weighted voting for replicated data. In *SOSP '79: Proceedings of the 7th ACM Symposium on Operating Systems Principles*, pages 150–162, New York, NY, USA, 1979. ACM.

[113] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, Washington, DC, USA, 2005. IEEE Computer Society.

[114] Patricia Gonzalez. MPI_Ckpt: Checkpoint and rollback-recovery library. Technical Report TR-001-2004, University of A Coruna, 2004.

[115] Christopher L. Gottbrath, Brian Barrett, Bill Gropp, Ewing "Rusty" Lusk, and Jeff Squyres. An interface to support the identification of dynamic MPI 2 processes for scalable parallel debugging. In *Proceedings of the 13th European PVM/MPI Users' Group Meeting*, Lecture Notes in Computer Science, Bonn, Germany, September 2006. Springer-Verlag.

[116] J. D. Gould. Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7(2):151–182, March 1975.

[117] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, New York, NY, USA, 1996. ACM.

[118] T. R. G. Green, M. E. Sime, and M. J. Fitter. The problems the programmer faces. *Ergonomics*, 23(9):893–907, September 1980.

[119] William Gropp and Ewing Lusk. Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[120] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[121] Indranil Gupta, Tushar D. Chandra, and German S. Goldszmidt. On scalable and efficient distributed failure detectors. In *PODC '01: Proceedings of the 20th annual ACM Symposium on Principles of Distributed Computing*, pages 170–179, New York, NY, USA, 2001. ACM Press.

[122] R. Gupta, P. Beckman, H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra. CIFTS: A coordinated infrastructure for fault-tolerant systems. *International Conference on Parallel Processing (ICPP)*, 2009.

[123] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated file system. In *USENIX Conference Proceedings*, pages 63–71, Anaheim, CA, June 1990. USENIX.

[124] Thomas J. Hacker, Fabian Romero, and Christopher D. Carothers. An analysis of clustered failures on large supercomputing systems. *Journal of Parallel and Distributed Computing*, 69(7):652–665, 2009.

[125] Jean-Michel Helary, Achour Mostefaoui, and Michel Raynal. Preventing useless checkpoints in distributed computations. In *SRDS '97: Proceedings of the 16th Symposium on Reliable Distributed Systems*, Washington, DC, USA, 1997. IEEE Computer Society.

[126] Jean-Michel Helary, Robert H. B. Netzer, and Michel Raynal. Consistency issues in distributed checkpoints. *IEEE Transactions on Software Engineering*, 25(2):274–281, 1999.

[127] Shang-Te Hsu and Ruei-Chuan Chang. Continuous checkpointing: Joining the checkpointing with virtual memory paging. *Software: Practice and Experience*, 27(9):1103–1120, 1997.

[128] Chao Huang. System support for checkpoint and restart of Charm++ and AMPI applications. Master's thesis, Dept. of Computer Science, University of Illinois, 2004.

[129] Chao Huang, Gengbin Zheng, and Laxmikant V. Kale. Supporting adaptivity in MPI for dynamic parallel applications. Technical Report 07-08, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 2007.

[130] Chao Huang, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. Performance evaluation of Adaptive MPI. In *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, March 2006.

[131] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528, 1984.

[132] Yennun Huang, P. Emerald Chung, Chandra Kintala, Chung-Yih Wang, and De-Ron Liang. NT-SwiFT: Software implemented fault tolerance on Windows NT. In *WINSYM'98: Proceedings of the 2nd Conference on USENIX Windows NT Symposium*, Berkeley, CA, USA, 1998. USENIX Association.

[133] Joshua Hursey, Chris January, Mark O'Connor, Paul H. Hargrove, David Lecomber, Jeffrey M. Squyres, and Andrew Lumsdaine. Checkpoint/restart-enabled parallel debugging. *Proceedings of the European MPI Users Group Conference (EuroMPI)*, September 2010.

[134] Joshua Hursey and Andrew Lumsdaine. A composable runtime recovery policy framework supporting resilient HPC applications. Under submission, 2010.

[135] Joshua Hursey, Timothy I. Mattox, and Andrew Lumsdaine. Interconnect agnostic checkpoint/restart in Open MPI. In *HPDC '09: Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing*, pages 49–58, New York, NY, USA, 2009. ACM.

[136] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, March 2007.

[137] Kai Hwang, Hai Jin, Edward Chow, Cho-Li Wang, and Zhiwei Xu. Designing SSI clusters with hierarchical checkpointing and single I/O space. *IEEE Concurrency*, 7(1):60–69, 1999.

[138] InfiniBand Trade Association. InfiniBand. `http://www.infinibandta.org`.

[139] K. A. Iskra, F. van der Linden, Z. W. Hendrikse, B. J. Overeinder, G. D. van Albada, and P. M. A. Sloot. The implementation of dynamite: An environment for migrating PVM tasks. *SIGOPS Operating Systems Review*, 34(3):40–55, 2000.

[140] G. (John) Janakiraman, Jose Renato Santos, and Dinesh Subhraveti. Cruz: Application-transparent distributed checkpoint-restart on standard operating systems. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 260–269, Washington, DC, USA, 2005. IEEE Computer Society.

[141] David P. Jasper. A discussion of checkpoint/restart. *Software Age*, pages 9–14, October 1969.

[142] Hai Jiang and Vipin Chaudhary. Process/thread migration and checkpointing in heterogeneous distributed systems. *Hawaii International Conference on System Sciences*, 9, 2004.

[143] Hai Jin and Kai Hwang. Distributed checkpointing on clusters with dynamic striping and staggering. In *ASIAN '02: Proceedings of the7th Asian Computing Science Conference on Advances in Computing Science*, pages 19–33, London, UK, 2002. Springer-Verlag.

[144] Hideyuki Jitsumoto, Toshio Endo, and Satoshi Matsuoka. ABARIS: An adaptable fault detection/recovery component framework for MPIs. *International Parallel and Distributed Processing Symposium*, 2007.

[145] David B. Johnson and Willy Zwaenepoel. Recovery in distributed systems using optimistic message logging and check-pointing. *Journal of Algorithms*, 11(3):462–491, 1990.

[146] P. W. Jones, P. H. Worley, Y. Yoshida, J. B. White, III, and J. Levesque. Practical performance portability in the parallel ocean program (POP): Research articles. *Concurrency and Computation: Practice & Experience*, 17(10):1317–1327, 2005.

[147] Hyungsoo Jung, Dongin Shin, Hyuck Han, Jai W Kim, Heon Y Yeom, and Jongsuk Lee. Design and implementation of multiple fault-tolerant MPI over Myrinet ($M^3$). *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, November 2005.

[148] F. Karablieh, R. Bazzi, and M. Hicks. Compiler-assisted heterogeneous checkpointing. *IEEE Symposium on Reliable Distributed Systems*, 2001.

[149] Youngbae Kim, James S. Plank, and Jack J. Dongarra. Fault tolerant matrix operations for networks of workstations using multiple checkpointing. *International Conference on High Performance Computing and Grid in Asia Pacific Region*, pages 460–465, 1997.

[150] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, Berkeley, CA, USA, 2005. USENIX Association.

[151] Richard Koo and Sam Toueg. Checkpointing and rollback-recovery for distributed systems. In *ACM '86: Proceedings of 1986 ACM Fall Joint Computer Conference*, pages 1150–1158, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[152] Dieter Kranzlmuller, Nam Thoai, and Jens Volkert. Debugging large-scale, long-running parallel programs. In *ICCS '02: Proceedings of the International Conference on Computational Science-Part II*, pages 913–922, London, UK, 2002. Springer-Verlag.

[153] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant stream processing using a distributed, replicated file system. *Proceedings of the VLDB Endowment*, 1(1):574–585, 2008.

[154] O. Laadan, D. Phung, and J. Nieh. Transparent checkpoint-restart of distributed applications on commodity clusters. *IEEE International Conference on Cluster Computing*, pages 1–13, 2005.

[155] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[156] H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman. Data replication strategies in grid environments. In *ICA3PP '02: Proceedings of the Fifth International Conference on Algorithms and Architectures for Parallel Processing*, Washington, DC, USA, 2002. IEEE Computer Society.

[157] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[158] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[159] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[160] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra. Recovery patterns for iterative methods in a parallel unstable environment. *SIAM Journal of Scientific Computing*, 30(1):102–116, 2007.

[161] T.J. Leblanc and J.M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[162] Troy LeBlanc, Rakhi Anand, Edgar Gabriel, and Jaspal Subhlok. VolpexMPI: an MPI library for execution of parallel applications on volatile nodes. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, September 2009.

[163] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 115–124, Washington, DC, USA, 2004. IEEE Computer Society.

[164] Claudia Leopold and Michael Sub. Observations on MPI-2 support for hybrid master/slave applications in dynamic and heterogeneous environments. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192, pages 285–292, September 2006.

[165] C. C. Li, S. K. Chen, W. K. Fuchs, and W. W. Hwu. Compiler-assisted multiple instruction retry. *IEEE Transactions on Computers*, 44(1), January 1995.

[166] Chung-Chi Jim Li, Elliot M. Stewart, and W. Kent Fuchs. Compiler-assisted full checkpointing. *Software: Practice and Experience*, 24(10):871–886, 1994.

[167] Wei-Jih Li and Jyh-Jong Tsay. Checkpointing message-passing interface (MPI) parallel programs. In *PRFTS '97: Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems*, Washington, DC, USA, 1997. IEEE Computer Society.

[168] Yawei Li and Zhiling Lan. Proactive fault manager for high performance computing. In *Proceedings of the International Conference on Dependable Systems and Networks (Fast Abstract)*, Yokohama, Japan, 2005.

[169] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the UNIX kernel. *Mobility: Processes, Computers, and Agents*, pages 154–162, 1999.

[170] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report 1346, University of Wisconsin-Madison, 1997.

[171] Hatem Ltaief, Edgar Gabriel, and Marc Garbey. Fault tolerant algorithms for heat transfer problems. *Journal of Parallel and Distributed Computing*, 68(5):663–677, 2008.

[172] Gerald Q. Maguire, Jr. and Jonathan M. Smith. Process migration: Effects on scientific computation. *SIGPLAN Notices*, 23(3):102–106, 1988.

[173] D. Manivannan, Robert H. B. Netzer, and Mukesh Singhal. Finding consistent global checkpoints in a distributed computation. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):623–627, 1997.

[174] D. Manivannan and Mukesh Singhal. Quasi-synchronous checkpointing: Models, characterization, and classification. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):703–713, 1999.

[175] Daniel Marques, Greg Bronevetsky, Rohit Fernandes, Keshav Pingali, and Paul Stodghil. Optimizing checkpoint sizes in the $C^3$ system. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium - Workshop 10*, Washington, DC, USA, 2005. IEEE Computer Society.

[176] M. Maruyama, T. Tsumura, and H. Nakashima. Parallel program debugging based on data-replay. In *Proceedings of the Parallel and Distributed Computing and Systems (PDCS)*, Phoenix, AZ, November 2005.

[177] John Mehnert-Spahn, Thomas Ropars, Michael Schoettner, and Christine Morin. The architecture of the XtreemOS grid checkpointing service. *Euro-Par 2009*, 2009.

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

[179] Barton P. Miller and Jong-Deok Choi. A mechanism for efficient debugging of parallel programs. *SIGPLAN Notices*, 24(1):141–150, 1989.

[180] Subhasish Mitra, Norbert Seifert, Ming Zhang, Quan Shi, and Kee Sup Kim. Robust system design with built-in soft-error resilience. *Computer*, 38:43–52, 2005.

[181] J. C. Mourino, M. J. Martin, P. Gonzalez, and R. Doallo. Fault-tolerant solutions for a MPI compute intensive application. In *PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 246–253, Washington, DC, USA, 2007. IEEE Computer Society.

[182] MPI Forum. MPI 3.0 Fault Tolerance Working Group. `http://meetings.mpi-forum.org/mpi3.0_ft.php`.

[183] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 115–128, Berkeley, CA, USA, 2004. USENIX Association.

[184] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive fault tolerance for HPC with Xen virtualization. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 23–32, New York, NY, USA, 2007. ACM.

[185] Hiroshi Nakamura, Takuro Hayashida, Masaaki Kondo, Yuya Tajima, Masashi Imai, and Takashi Nanya. Skewed checkpointing for tolerating multi-node failures. In *SRDS '04: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 116–125, Washington, DC, USA, 2004. IEEE Computer Society.

[186] Hyo-chang Nam, Jong Kim, SungJe Hong, and Sunggu Lee. Probabilistic checkpointing. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, Washington, DC, USA, 1997. IEEE Computer Society.

[187] National Aeronautics and Space Administration. NAS parallel benchmarks. `http://www.nas.nasa.gov/Resources/Software/npb.html`.

[188] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. *Journal of Supercomputing*, 8(4):371–388, 1995.

[189] Robert H. B. Netzer, Sairam Subramanian, and Jian Xu. Critical-path-based message logging for incremental replay of message-passing programs. Technical Report CS-94-19, Brown University, Providence, RI, USA, 1994.

[190] Robert H. B. Netzer and Jian Xu. Adaptive message logging for incremental program replay. *IEEE Parallel Distributed Technology*, 1(4):32–39, 1993.

[191] Robert H. B. Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, 1995.

[192] Robert H. B. Netzer and Yikang Xu. Replaying distributed programs without message logging. In *HPDC '97: Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, Washington, DC, USA, 1997. IEEE Computer Society.

[193] N. Neves and W. K. Fuchs. RENEW: A tool for fast and efficient implementation of checkpoint protocols. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, Washington, DC, USA, 1998. IEEE Computer Society.

[194] Anh Nguyen-Tuong and Andrew S. Grimshaw. Using reflection for incorporating fault-tolerance techniques into distributed applications. Technical Report CS-98-34, University of Virginia, Charlottesville, VA, USA, 1998.

[195] Ron Oldfield. Investigating lightweight storage and overlay networks for fault tolerance. *High Availability and Performance Computing Workshop 2006 (HAPCW06)*, October 2006.

[196] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *MSST '07: Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[197] Adam J. Oliner, Larry Rudolph, and Ramendra K. Sahoo. Cooperative checkpointing: A robust approach to large-scale systems reliability. In *ICS '06: Proceedings of the 20th Annual International Conference on Supercomputing*, pages 14–23, New York, NY, USA, 2006. ACM.

[198] Adam J. Oliner, R. K. Sahoo, J. E. Moreira, and M. Gupta. Performance implications of periodic checkpointing on large-scale cluster systems. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium - Workshop 18*, Washington, DC, USA, 2005. IEEE Computer Society.

[199] Adam J. Oliner and Jon Stearley. What supercomputers say: A study of five system logs. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 575–584, Washington, DC, USA, 2007. IEEE Computer Society.

[200] OpenFabrics Alliance. OpenFabrics. `http://www.openfabrics.org/`.

[201] Douglas Z. Pan and Mark A. Linton. Supporting reverse execution for parallel programs. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, New York, NY, USA, 1988. ACM Press.

[202] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD '88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, New York, NY, USA, 1988. ACM.

[203] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL: A portable implementation of the High-Performance Linpack benchmark for distributed-memory computers, September 2008.

[204] Fabrizio Petrini, Jarek Nieplocha, and Vinod Tipparaju. SFT: Scalable fault tolerance. *SIGOPS Operating Systems Review*, 40(2):55–62, 2006.

[205] J. S. Plank. Improving the performance of coordinated checkpointers on networks of workstations using RAID techniques. In *SRDS '96: Proceedings of the 15th Symposium on Reliable Distributed Systems*, Washington, DC, USA, 1996. IEEE Computer Society.

[206] J. S. Plank and W. R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, Washington, DC, USA, 1998. IEEE Computer Society.

[207] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.

[208] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[209] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. In *Software – Practice and Experience*, volume 29, pages 125–142, 1999.

[210] James S. Plank and Kai Li. ickp: A consistent checkpointer for multicomputers. *IEEE Concurrency*, 2(2):62–67, 1994.

[211] Steve J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *Journal of Computational Physics*, 117:1–19, 1995.

[212] Dhiraj K. Pradhan and Nitin H. Vaidya. Roll-forward and rollback recovery: Performance-reliability trade-off. *IEEE Transactions on Computers*, 46(3):372–378, 1997.

[213] D.K. Pradhan and N.H Vaidya. Roll-forward checkpointing scheme: A novel fault-tolerant architecture. *IEEE Transactions on Computers*, 43(10):1163–1174, 1994.

[214] Jim Pruyne and Miron Livny. Managing checkpoints for parallel programs. In *IPPS '96: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 140–154, London, UK, 1996. Springer-Verlag.

[215] Milos Prvulovic, Zheng Zhang, and Josep Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. *SIGARCH Computer Architecture News*, 30(2):111–122, 2002.

[216] C. Pu, J.D. Noe, and A. Proudfoot. Regeneration of replicated objects: A technique and its Eden implementation. *IEEE Transactions on Software Engineering*, 14(7):936–945, 1988.

[217] Sasikumar Punnekkat, Alan Burns, and Robert Davis. Analysis of checkpointing for real-time systems. *Real-Time Systems*, 20(1):83–102, 2001.

[218] Francesco Quaglia. Combining periodic and probabilistic checkpointing in optimistic simulation. In *PADS '99: Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 109–116, Washington, DC, USA, 1999. IEEE Computer Society.

[219] B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, New York, NY, USA, 1975. ACM Press.

[220] Sridharan Ranganathan, Alan D. George, Robert W. Todd, and Matthew C. Chidester. Gossip-style failure detection and distributed consensus for scalable heterogeneous clusters. *Cluster Computing*, 4(3):197–209, 2001.

[221] Sriram Rao, Lorenzo Alvisi, and Harrick 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*, Washington, DC, USA, 1999. IEEE Computer Society.

[222] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. The cost of recovery in message logging protocols. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):160–173, 2000.

[223] Daniel A. Reed, Charng da Lu, and Celso L. Mendes. Reliability challenges in large systems. In *Future Generation Computer Systems*, volume 22, pages 293–302, February 2006.

[224] Michael Richmond and Michael Hitchens. A new process migration algorithm. *SIGOPS Operating Systems Review*, 31(1):31–42, 1997.

[225] Pierre Riteau, Adrien Lebre, and Christine Morin. Handling persistent states in process checkpoint/restart mechanisms for HPC systems. *IEEE International Symposium on Cluster Computing and the Grid*, May 2009.

[226] G. Rodriguez, M.J. Martin, P. Gonzalez, J. Tourino, and R. Doallo. CPPC: A compiler-assisted tool for portable checkpointing of message-passing applications. *Proceedings of the International Workshop on Scalable Tools for High-End Computing (STHEC 2008)*, pages 1–12, June 2008.

[227] M.A. Ronsse and D.A. Kranzlmüller. RoltMP-replay of Lamport timestamps for message passing systems. *Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, 1998.

[228] Michiel Ronsse, Koen De Bosschere, and Jacques Chassin de Kergommeaux. Execution replay and debugging. In *Proceedings of the Fourth International Workshop on Automated Debugging (AADebug)*, Munich, Germany, August 2000.

[229] Michiel Ronsse, Mark Christiaens, and Koen De Bosschere. Debugging shared memory parallel programs using record/replay. *Future Generation Computer Systems*, 19(5):679–687, 2003.

[230] Joseph Ruscio, Michael Heffner, and Srinidhi Varadarajan. DejaVu: Transparent user-level checkpointing, migration and recovery for distributed systems. *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, November 2006.

[231] Sang-Moon Ryu and Dong-Jo Park. Checkpointing for the reliability of real-time systems with on-line fault detection. *Embedded and Ubiquitous Computing*, 3824, 2005.

[232] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *KDD '03: Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 426–435, New York, NY, USA, 2003. ACM.

[233] 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.

[234] 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.

[235] D.P. Scarpazza, P. Mullaney, O. Villa, F. Petrini, V. Tipparaju, D.M.L. Brown, and J. Nieplocha. Transparent system-level migration of PGAS applications using Xen on InfiniBand. *IEEE International Conference on Cluster Computing*, pages 74–83, 2007.

[236] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78, 2007.

[237] A. David Selvakumar, P. M. Sobha, G. C. Ravindra, and R. Pitchiah. Design, implementation and performance of fault-tolerant message passing interface (MPI). In *Proceedings of the Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region (HPCASIA '04)*, pages 120–129, Washington, DC, USA, 2004. IEEE Computer Society.

[238] L. M. Silva, J. G. Silva, S. Chapple, and L. Clarke. Portable checkpointing and recovery. In *Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing (HPDC '95)*, Washington, DC, USA, 1995. IEEE Computer Society.

[239] L.M. Silva and J.G. Silva. System-level versus user-defined checkpointing. *IEEE Symposium on Reliable Distributed Systems*, 1998.

[240] Luis Moura Silva and Joao Gabriel Silva. The performance of coordinated and independent checkpointing. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 280–284, Washington, DC, USA, 1999. IEEE Computer Society.

[241] Jonathan M. Smith. A survey of process migration mechanisms. *SIGOPS Operating Systems Review*, 22(3):28–40, 1988.

[242] Peter Sobe. Stable checkpointing in distributed systems without shared disks. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, Washington, DC, USA, 2003. IEEE Computer Society.

[243] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, and David A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. *SIGARCH Computer Architecture News*, 30(2):123–134, 2002.

[244] Carlos Sosa. IBM system Blue Gene solution: Blue Gene/P application development. Technical report, IBM, December 2008.

[245] Jeffrey M. Squyres and Andrew Lumsdaine. The component architecture of Open MPI: Enabling third-party collective algorithms. In *Proceedings of the 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, pages 167–185, St. Malo, France, July 2004. Springer.

[246] Johny Srouji, Paul Schuster, maury Bach, and Yulik Kuzmin. A transparent checkpoint facility on NT. In *Proceedings of the 2nd USENIX Windows NT Symposium*. USENIX, August 1998.

[247] Kuo-Feng Ssu, Bin Yao, and W. Kent Fuchs. An adaptive checkpointing protocol to bound recovery time with message logging. *IEEE Symposium on Reliable Distributed Systems*, page 244, 1999.

[248] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *10th International Parallel Processing Symposium (IPPS '96)*, 1996.

[249] Michael Stonebraker and Gerhard A. Schloss. Distributed RAID - a new multiple copy algorithm. In *Proceedings of the Sixth International Conference on Data Engineering*, pages 430–437, Washington, DC, USA, 1990. IEEE Computer Society.

[250] Rob Strom and Shaula Yemini. Optimistic recovery in distributed systems. *ACM Transactions Computer Systems*, 3(3):204–226, 1985.

[251] Volker Strumpen. Portable and fault-tolerant software systems. *IEEE Micro*, 18(5):22–32, 1998.

[252] Rajagopal Subramaniyan, Eric Grobelny, Scott Studham, and Alan D. George. Optimization of checkpointing-related I/O for high-performance parallel and distributed computing. *Journal of Supercomputing*, 46(2):150–180, 2008.

[253] Hiroyuki Takizawa, Katsuto Sato, Kazuhiko Komatsu, and Hiroaki Kobayashi. CheCUDA: A checkpoint/restart tool for CUDA applications. In *PDCAT '09: Proceedings of the 2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 408–413, Washington, DC, USA, 2009. IEEE Computer Society.

[254] C. P. Tan, Weng-Fai Wong, and Chung-Kwong Yuen. tmPVM - task migratable PVM. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 196–202, Washington, DC, USA, 1999. IEEE Computer Society.

[255] Yuan Tang, Graham E. Fagg, and Jack J. Dongarra. Proposal of MPI operation level checkpoint/rollback and one implementation. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 27–34, Washington, DC, USA, 2006. IEEE Computer Society.

[256] Asser N. Tantawi and Manfred Ruschitzka. Performance analysis of checkpointing strategies. *ACM Transactions Computer Systems*, 2(2):123–144, 1984.

[257] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable remote execution facilities for the V-system. *SIGOPS Operating Systems Review*, 19(5):2–12, 1985.

[258] Nam Thoai, Dieter Kranzlmuller, and Jens Volkert. ROS: The rollback-one-step method to minimize the waiting time during debugging long-running parallel programs. *High Performance Computing for Computational Science (VECPAR 2002)*, 2565:87–94, 2002.

[259] TotalView Technologies. ReplayEngine, 2009.

[260] Undo Ltd. UndoDB - Reversible debugging for Linux, 2009.

[261] Sathish S. Vadhiyar and Jack J. Dongarra. A performance oriented migration framework for the grid. *IEEE International Symposium on Cluster Computing and the Grid*, 2003.

[262] Sathish S. Vadhiyar and Jack J. Dongarra. SRS - a framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[263] Nitin H Vaidya. A case for two-level distributed recovery schemes. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*, pages 64–73, 1995.

[264] Nitin H. Vaidya. On staggered checkpointing. In *SPDP '96: Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP '96)*, Washington, DC, USA, 1996. IEEE Computer Society.

[265] Nitin H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Transactions on Computers*, 46(8):942–947, 1997.

[266] Nitin H Vaidya. Staggered consistent checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):694–702, 1999.

[267] Geoffroy Vallee, Kulathep Charoenpornwattana, Christian Engelmann, Anand Tikotekar, Chokchai Leangsuksun, Thomas Naughton, and Stephen L. Scott. A framework for proactive fault tolerance. *International Conference on Availability, Reliability and Security*, pages 659–664, 2008.

[268] Geoffroy Vallee, Renaud Lottiaux, David Margery, and Christine Morin. Ghost process: a sound basis to implement process duplication, migration and checkpoint/restart in Linux clusters. *International Symposium on Parallel and Distributed Computing*, pages 97–104, 2005.

[269] Geoffroy Vallee, Christine Morin, Renaud Lottiaux, Jean-Yves Berthou, and Ivan Dutka Malen. Process migration based on Gobelins distributed shared memory. *IEEE International Symposium on Cluster Computing and the Grid*, 2002.

[270] David Van Der Spoel, Eric Lindahl, Berk Hess, Gerrit Groenhof, Alan E. Mark, and Herman J. Berendsen. GROMACS: Fast, flexable, and free. *Journal of Computational Chemistry*, 26(16):1701–1718, 2005.

[271] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-based failure detection service. In *Middleware'98, IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 55–70, England, September 1998.

[272] VirtuTech. Reversible execution and debugging. Technical report, VirtuTech, June 2007.

[273] John Paul Walters and Vipin Chaudhary. A fault-tolerant strategy for virtualized HPC clusters. *Journal of Supercomputing*, December 2008.

[274] John Paul Walters and Vipin Chaudhary. Replication-based fault tolerance for MPI applications. *IEEE Transactions on Parallel and Distributed Systems*, 20(7):997–1010, 2009.

[275] John Paul Walters, Vipin Chaudhary, Minsuk Cha, Salvatore Guercio Jr., and Steve Gallo. A comparison of virtualization technologies for HPC. In *AINA '08: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications*, pages 861–868, Washington, DC, USA, 2008. IEEE Computer Society.

[276] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. *International Parallel and Distributed Processing Symposium*, 2007.

[277] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen L. Scott. Proactive process-level live migration in HPC environments. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[278] Yi-Min Wang. Consistent global checkpoints that contain a given set of local checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, 1997.

[279] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, Pe-Yu Chung, and C. Kintala. Checkpointing and its applications. In *FTCS '95: Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, Washington, DC, USA, 1995. IEEE Computer Society.

[280] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[281] Larry Wittie. The Bugnet distributed debugging system. In *EW 2: Proceedings of the 2nd Workshop on Making Distributed Systems Work*, pages 1–3, New York, NY, USA, 1986. ACM.

[282] Larry D. Wittie. Debugging distributed C programs by real time reply. In *PADD '88: Proceedings of the 1988 ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 57–67, New York, NY, USA, 1988. ACM.

[283] Namyoon Woo, Soonho Choi, Hyungsoo Jung, Jungwhan Moon, Heon Y. Yeom, Taesoon Park, and Hyungwoo Park. MPICH-GF: Providing fault tolerance on grid environments. *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid2003)*, 2003.

[284] Timothy S. Woodall, Galen M. Shipman, George Bosilca, Richard L. Graham, and Arthur 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.

[285] John W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, 1974.

[286] F. Zambonelli. On the effectiveness of distributed checkpoint algorithms for domino-free recovery. *International Symposium on High-Performance Distributed Computing*, 1998.

[287] Franco Zambonelli. An efficient logging algorithm for incremental replay of message. In *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, pages 392–398, Washington, DC, USA, 1999. IEEE Computer Society.

[288] E. Zayas. Attacking the process migration bottleneck. *SIGOPS Operating Systems Review*, 21(5):13–24, 1987.

[289] M. V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16(9), 1973.

[290] Yanyong Zhang, Mark S. Squillante, Anand Sivasubramaniam, and Ramendra K. Sahoo. Performance implications of failures in large-scale cluster scheduling. In *Job Scheduling Strategies for Parallel Processing*, volume 3277, pages 233–252, May 2004.

[291] Gengbin Zheng, Chao Huang, and Laxmikant V. Kalé. Performance evaluation of automatic checkpoint-based fault tolerance for AMPI and Charm++. *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April 2006.

[292] Qilong Zheng, Guoliang Chen, and Liusheng Huang. Optimal record and replay for debugging of non-deterministic MPI/PVM programs. *International Conference on High-Performance Computing in the Asia-Pacific Region*, 1:473, 2000.

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

[294] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *Science*, 206(4420):776–788, November 1979.

[295] Avi Ziv and Jehoshua Bruck. An on-line algorithm for checkpoint placement. *IEEE Transactions on Computers*, 46(9):976–985, 1997.

# A

# Checkpoint/Restart Application Programming Interface

Chapter 5 presented a variety of non-MPI-standard Application Programming Interfaces (APIs) that are available to the application as part of Open MPI's Extended Interfaces. This appendix provides more details and examples regarding these APIs.

## 1. Checkpoint/Restart Interface

The Checkpoint/Restart (C/R) API allows an MPI application to request a checkpoint or restart the application from within the program. Unless otherwise specified, these calls are collective over all processes in the MPI application.

**1.1. Interface.** The application can request a checkpoint of the application by using the OMPI_CR_CHECKPOINT API seen below. The OMPI_CR_CHECKPOINT returns a handle to the global snapshot reference and a unique sequence number. No info keywords are defined at this time.

| OMPI_CR_CHECKPOINT(handle, seq, info) | | |
|---|---|---|
| *OUT* | handle | Global snapshot reference (string) |
| *OUT* | seq | Sequence number (int) |
| *INOUT* | info | A set of key−value pairs providing additional information to the MPI implementation regarding how to continue after quiescence (handle, significant on all ranks) |

int OMPI_CR_CHECKPOINT(char **handle, int *seq, MPI_Info info);

Additionally the application can use the OMPI_CR_RESTART API to request a restart of the application, also seen below. The OMPI_CR_RESTART function takes as arguments the handle and sequence number returned by the OMPI_CR_CHECKPOINT or an external tool like `ompi-checkpoint`. The restart command is not collective and may be called by any process. The C/R infrastructure will act on each restart request in the order received. No info keywords are defined at this time.

| OMPI_CR_RESTART(handle, seq, info) | | |
|---|---|---|
| *IN* | handle | Global snapshot reference (string) |
| *IN* | seq | Sequence number (int) |
| *INOUT* | info | A set of key−value pairs providing additional information to the MPI implementation regarding how to continue after quiescence (handle, significant on all ranks) |

int OMPI_CR_RESTART(char *handle, int seq, MPI_Info info);

**1.2. Examples.** This section presents some pseudo code examples of how the C/R interfaces might be used in an MPI application.

1.2.1. *Example 1: Application Directed Checkpoint.* The application identifies a good point in the execution to checkpoint. The application may optionally use CRS specific interfaces to identify temporary buffers that may be excluded from the checkpoint.

```
#include <mpi.h>
#ifdef OPEN_MPI
  #include <mpi-ext.h>
#endif
{ MPI_Init(argc, argv);
  for(i=0; i < max_iter; ++i) {
#ifdef OMPI_HAVE_MPI_EXT_CR
    // Request a checkpoint before every step
    OMPI_CR_Checkpoint(&handle, &seq, MPI_INFO_NULL);
#endif
    // Resume normal operation.
  }
}
```

1.2.2. *Example 2: Application Directed Restart.* The application identifies a possible problem with one of the cooperating processes, and requests a restart of the entire application. The application may have noticed erroneous results from a peer (possibly indicating the effect of a soft-error) or received notification of a process failure from the MPI interface (e.g., MPI_SEND failed due to process loss).

```
#include <mpi.h>
#ifdef OPEN_MPI
  #include <mpi-ext.h>
#endif
{ MPI_Init(argc, argv);
  for(i=0; i < max_iter; ++i) {
#ifdef OMPI_HAVE_MPI_EXT_CR
    // Request a checkpoint before every step
    OMPI_CR_Checkpoint(&handle, &seq);
#endif
    // Resume normal operation.
    if( MPI_SUCCESS != MPI_Send(...) ) {
#ifdef OMPI_HAVE_MPI_EXT_CR
      // Restart from the last checkpoint, and keep processing
      OMPI_CR_Restart(handle, seq, MPI_INFO_NULL);
#else
      MPI_Abort(MPI_COMM_WORLD, -1);
```

```
#endif
    }
  }
}
```

## 2. Quiescence Interface

Parallel applications can benefit from having a synchronization call that synchronizes all outstanding communication on a communicator. The quiesce operation on communicators serves a similar purpose as MPI_WIN_FENCE does to one-sided RMA operations on a *window*. The quiescence collective operations allow the application to force all outstanding communication to the intended recipient of the communication.

A region of quiescence is defined between the start and end calls. This allows the application to define a region of time where they are assured that no communication occurs over the designated communicator. This can be useful when checkpointing an application, or preparing buffer space for additional communication.

**2.1. Interface.** The interface to the blocking version of OMPI_CR_QUIESCE_START operation is below. Though a communicator argument is provided, it is required to be MPI_COMM_WORLD in the prototype implementation in Open MPI. No info keywords are defined at this time.

```
OMPI_CR_QUIESCE_START(comm, info)
 IN      comm    communicator (handle)
 INOUT   info    A set of key−value pairs providing hints to the MPI
                 implementation regarding how this function should
                 behave (handle, significant on all ranks)
int OMPI_CR_QUIESCE_START(MPI_Comm comm, MPI_Info info);
```

Below is the interface to the blocking version of the OMPI_CR_QUIESCE_END operation. Though a communicator argument is provided, the prototype implementation in Open MPI requires it to be MPI_COMM_WORLD. No info keywords are defined at this time.

| OMPI_CR_QUIESCE_END(comm, info) | | |
|---|---|---|
| *IN* | comm | communicator (handle) |
| *INOUT* | info | A set of key−value pairs providing additional information to the MPI implementation regarding how to continue after quiescence (handle, significant on all ranks) |

int OMPI_CR_QUIESCE_END(MPI_Comm comm, MPI_Info info);

The OMPI_CR_QUIESCENCE_CHECKPOINT function allows the application to optionally choose to use the MPI provided CRS. This interface assumes that the MPI library has been quiesced by a previous call to the quiescence start function, in contrast to the OMPI_CR_CHECKPOINT API (presented in Section 1) which has no such requirement. Though a communicator argument is provided, it is required to be MPI_COMM_WORLD in the prototype implementation in Open MPI. No info keywords are defined at this time.

| OMPI_CR_QUIESCE_CHECKPOINT(comm, handle, seq, info) | | |
|---|---|---|
| *IN* | comm | communicator (handle) |
| *OUT* | handle | Global snapshot reference (string ) |
| *OUT* | seq | Sequence number (int) |
| *INOUT* | info | A set of key−value pairs providing hints to the MPI implementation regarding how this function should behave (handle, significant on all ranks) |

int OMPI_CR_QUIESCE_CHECKPOINT(MPI_Comm comm, char **handle, int *seq, MPI_Info info);

**2.2. Examples.** Next we present some pseudo code examples of how the quiescence interfaces might be used in MPI applications.

2.2.1. *Example 1: Application-Level Checkpoint/Restart.* The application requires assurance that all sent messages have been either received by or cached by the recipient on the communicator before taking an application-level checkpoint. On restart, the application is responsible for distributing the checkpoint data, and, if necessary, specifying to the application that it is restarting.

```
#include <mpi.h>
#ifdef OPEN_MPI
  #include <mpi-ext.h>
#endif
{ MPI_Init(argc, argv);
#ifdef OMPI_HAVE_MPI_EXT_CR
  OMPI_CR_Quiesce_start(MPI_COMM_WORLD, MPI_INFO_NULL);
  // Prepare application for application-level checkpoint.
  // Wait on any important outstanding receives
  // Save application state
  OMPI_CR_Quiesce_end(MPI_COMM_WORLD, MPI_INFO_NULL);
#endif
  // Resume normal operation.
}
```

2.2.2. *Example 2: Application Controlled System-Level Checkpoint/Restart.* The application requires the ability to perform action during the quiescent region before or after the system-level checkpoint is requested from the MPI implementation provided CRS. For example, the application may want to highlight temporary buffers that should not be preserved in the checkpoint to reduce the size of the checkpoint.

```
#include <mpi.h>
#ifdef OPEN_MPI
  #include <mpi-ext.h>
#endif
{ MPI_Init(argc, argv);
#ifdef OMPI_HAVE_MPI_EXT_CR
  OMPI_CR_Quiesce_start(MPI_COMM_WORLD, MPI_INFO_NULL);
  // Prepare application for checkpoint.
  // Wait on any important outstanding receives
  // Mark some memory regions for exclusion
  OMPI_CR_Quiesce_checkpoint(MPI_COMM_WORLD, &handle, &seq,
                             MPI_INFO_NULL);
  OMPI_CR_Quiesce_end(MPI_COMM_WORLD, MPI_INFO_NULL);
#endif
  // Resume normal operation.
}
```

## 3. Process Migration Interface

The process migration API allows an MPI application to migrate processes defined by a communicator, possibly away from their current resources. The MPI application may do so to avoid a predicted node failure or to better load balance their application.

**3.1. Interface.** The application can request a migration of the application by using the OMPI_CR_MIGRATE API seen below. The group of migrating processes is defined by the communicator passed. The application can optionally specify a hostname or rank to move the current rank onto or close to. If the `CR_OFF_NODE` MPI_INFO keyword to `true` then the process is moved away from the current host to a spare or sparsely loaded host. The operation is collective across the communicator provided.

| | | |
|---|---|---|
| OMPI_CR_MIGRATE(comm, hostname, rank, info) | | |
| *IN* | comm | Communicator of processes to migrate |
| *IN* | hostname | Name of the machine to move this rank onto. May be NULL. (string) |
| *IN* | rank | Process rank to move this rank close to. May be negative, indicating NULL. (int) |
| *INOUT* | info | A set of key−value pairs providing hints to the MPI implementation regarding how this function should behave (handle, significant on all ranks) |
| int OMPI_CR_MIGRATE(MPI_Comm comm, char *hostname, int rank, MPI_Info info) | | |

**3.2. Examples.** This section presents some pseudo code examples of how the process migration interfaces might be used in MPI applications.

3.2.1. *Example 1: Application Directed Fault Avoidance.* Individual ranks in the application receive external notification that their machine is going to fail in the near future. The processes can then use previously created communicators to migrate, or individually choose to move using MPI_COMM_SELF.

```
#include <mpi.h>
#ifdef OPEN_MPI
  #include <mpi-ext.h>
#endif
{ MPI_Info qinfo;
  MPI_Init(argc, argv);

  for(i=0; i < max_iter; ++i) {
    // Receive notification that this node is going to fail
#ifdef OMPI_HAVE_MPI_EXT_CR
    // Asked to be migrated anywhere else in the system,
    // except this node.
    MPI_Info_set(qinfo, "CR_OFF_NODE", "true");
    OMPI_CR_MIGRATE(MPI_COMM_SELF, NULL, -1, MPI_INFO_NULL);
#endif
    // Resume normal operation.
  }
}
```

3.2.2. *Example 2: Application Directed Load Balancing.* In Figure A.1 the application identifies a section of time when it wishes to reposition the processes in the computation to reduce message delay. This is useful for applications that have multiple distinct phases of computation with different communication patterns.

## 4. Interlayer Notification Callback Callbacks

Figure A.2 presents the INC registration functions. Table A.1 presents the various arguments passed to these callback functions. Applications can use these callbacks to synchronize additional libraries (e.g., accelerators [**253**]) or files that would not otherwise be accounted for in the local snapshot. The INC callbacks are called before and after the underlying MPI library's callbacks. This allows the application to use MPI function to coordinate processes during checkpoint preparation and upon recovery, if necessary.

```
#include <mpi.h>
#ifdef OPEN_MPI
  #include <mpi-ext.h>
#endif
{ MPI_Init(argc, argv);
  ...
  // Stage 1: Communication Pattern A
  for(i=0; i < max_iter; ++i) {
    ...
  }
#ifdef OMPI_HAVE_MPI_EXT_CR
  // Since the communication pattern is changing,
  // re-position my processes by using process migration.
  neighbor_rank = get_best_neighbor(my_rank);
  OMPI_CR_MIGRATE(MPI_COMM_WORLD, NULL, neighbor_rank,
                  MPI_INFO_NULL);
#endif
  // Stage 2: Communication Pattern B
  for(i=0; i < max_iter; ++i) {
    ...
  }
}
```

FIGURE A.1. Process Migration API Example.

```
// INC Registration Function
int OMPI_CR_INC_register_callback(OMPI_CR_INC_callback_event_t event,
                                  OMPI_CR_INC_callback_function function,
                                  OMPI_CR_INC_callback_function *prev_function);
// INC Callback Function Signature
typedef int (*OMPI_CR_INC_callback_function)(OMPI_CR_INC_callback_event_t event,
                                             OMPI_CR_INC_callback_state_t state);
```

FIGURE A.2. Open MPI INC Registration API.

| **OMPI_CR_INC_callback_event_t** | |
|---|---|
| **State** | **Description** |
| OMPI_CR_INC_PRE_CRS_PRE_MPI | Pre-checkpoint, before OMPI INC. |
| OMPI_CR_INC_PRE_CRS_POST_MPI | Pre-checkpoint, after OMPI INC. |
| OMPI_CR_INC_POST_CRS_PRE_MPI | Continue/Restart, before OMPI INC. |
| OMPI_CR_INC_POST_CRS_POST_MPI | Continue/Restart, after OMPI INC. |
| **OMPI_CR_INC_callback_state_t** | |
| **State** | **Description** |
| OMPI_CR_INC_STATE_PREPARE | Pre-checkpoint |
| OMPI_CR_INC_STATE_CONTINUE | Continue |
| OMPI_CR_INC_STATE_RESTART | Restart |
| OMPI_CR_INC_STATE_ERROR | Error |

TABLE A.1. Open MPI INC function callback events and states.

# B

# Command Line Tools

This chapter describes the command line tools: `ompi-checkpoint`, `ompi-restart` and `ompi-migrate`. These commands are used by an end user (e.g., scheduler, resource manager, system administrator, developer) to interact with the C/R functionality in Open MPI from the command line. These commands transparently activate the appropriate C/R functionality in the running MPI application.

### 1. `ompi-checkpoint`

The `ompi-checkpoint` command is provided to checkpoint an MPI application. The one required argument to this command is the PID of the `mpirun` process. This command must be launched on the same machine as the running `mpirun` process. Once a checkpoint request has completed `ompi-checkpoint` will return a global snapshot reference and a sequence

```
shell$ ompi−checkpoint PID_OF_MPIRUN [OPTIONS]
shell$ ompi−restart GLOBAL_SNAPSHOT_REF [OPTIONS]
shell$ ompi−migrate PID_OF_MPIRUN [OPTIONS]
```

FIGURE B.1. Open MPI `ompi-checkpoint`, `ompi-restart`, and `ompi-migrate` commands.

| Argument | Description |
|---|---|
| `PID_OF_MPIRUN` | PID of the mpirun process |
| `-h` `--help` | Display help |
| `-v` `--verbose` | Display verbose output |
| `-V #` | Display verbose output up to a specified level |
| `--term` | Terminate the application after checkpoint. |
| `-s` `--status` | Display status progression messages of the checkpoint. |
| `-l` `--list` | Display a list of checkpoint files available on this machine |
| `--stop` | Send `SIGSTOP` to application just after checkpoint. |
| `--detach` | Do not wait for debugger to re-attach after a checkpoint. |
| `--attach` | Wait for debugger to attach after a checkpoint. |

TABLE B.1. Open MPI `ompi-checkpoint` arguments.

number. This information will allow the end user to properly restart the MPI application at a later time.

End users familiar with the LAM/MPI checkpoint/restart commands should notice that the `ompi-checkpoint` does not require the user to tell it which CRS (e.g., BLCR or SELF) to use when checkpointing the application. This information is automatically detected and stored with the checkpoint snapshots.

**1.1. Interface.** Figure B.1 presents the interface to the `ompi-checkpoint` command. Table B.1 explains the command line arguments for this command.

**1.2. Example.** Figure B.2 presents a brief example of how the `ompi-checkpoint` command could be used to checkpoint a running MPI application. The checkpoint command generated two checkpoints with sequence numbers `0` and `1`, and a global snapshot reference of `ompi-global-snapshot-1234`.

```
shell$ mpirun my-app <args> &
shell$ export PID_OF_MPIRUN=1234
shell$ ompi-checkpoint $PID_OF_MPIRUN
Snapshot Ref.: 0 ompi-global-snapshot-1234
shell$ ompi-checkpoint $PID_OF_MPIRUN
Snapshot Ref.: 1 ompi-global-snapshot-1234
```

FIGURE B.2. Open MPI `ompi-checkpoint` example.

| Argument | Description |
|---|---|
| `GLOBAL_SNAPSHOT_REF` | Global snapshot reference |
| `-h` \| `--help` | Display help |
| `-v` \| `--verbose` | Display verbose output |
| `-a` \| `--apponly` | Only create the app context file, do not restart from it. |
| `-s` \| `--seq` | The sequence number of the checkpoint to start from. (Default: -1, or most recent) |
| `--hostfile` \| `--machinefile` | Provide a hostfile to use for launch. |
| `-i` \| `--info` | Display information about the checkpoint |
| `--mpirun_opts` | Options to pass directly to `mpirun` |
| `--crdebug` | Restart and wait for the debugger to attach. |

TABLE B.2. Open MPI `ompi-restart` arguments.

### 2. `ompi-restart`

The `ompi-restart` command is provided to restart a previously-checkpointed MPI application. The one required argument to this command is the global snapshot reference returned by `ompi-checkpoint`. The global snapshot reference contains all of the necessary information to properly restart an MPI application. Invoking `ompi-restart` results in a new `mpirun` being `exec()`'ed in its place.

End users familiar with the LAM/MPI checkpoint/restart commands should notice that the `ompi-restart` does not require the user to tell it which CRS (e.g., BLCR or SELF) was used when checkpointing the application. This information is stored with the checkpoint snapshot and automatically used by the `ompi-restart` command.

**2.1. Interface.** Figure B.1 presents the interface to the `ompi-restart` command. Table B.2 explains the command line arguments.

| Argument | | Description |
|---|---|---|
| PID_OF_MPIRUN | | PID of the mpirun process |
| -h | --help | Display help |
| -v | --verbose | Display verbose output |
| -x | --off | List of nodes to migrate off of (comma separated). |
| -r | --ranks | List of MPI_COMM_WORLD ranks to migrate (comma separated). |
| -t | --onto | List of nodes to migrate onto (comma separated). |

TABLE B.3. Open MPI ompi-migrate arguments.

**2.2. Example.** Below is a brief example of how the ompi-restart command could be used to restart a previously checkpointed MPI application. ompi-global-snapshot-1234 is the global snapshot reference returned by ompi-checkpoint, this checkpoint has 2 sequence numbers. The first example restarts from the most recent sequence number (1). The second example restarts from the first sequence number (0).

```
shell$ ompi-restart ompi-global-snapshot-1234
shell$ ompi-restart -s 0 ompi-global-snapshot-1234
```

### 3. ompi-migrate

The ompi-migrate command is provided to migrate, using the C/R infrastructure, a group of MPI processes in a running application. The two required arguments to this command are the PID of the mpirun process, and the hosts or MPI_COMM_WORLD ranks to migrate. Optionally, the end user can specify a destination set of resources. This command must be launched on the same machine as the running mpirun process.

**3.1. Interface.** Figure B.1 presents the interface to the ompi-migrate command. Table B.3 explains the command line arguments for this command.

**3.2. Example.** Below is a brief example of how the ompi-migrate command could be used to migrate a C/R enabled MPI application. In this example, we are requesting that all processes be migrated off of node1 and node2.

```
shell$ ompi-migrate $PID_OF_MPIRUN --off node1,node2
```

# C

# self CRS

The self Checkpoint/Restart Service component will invoke user-defined functions to save and restore checkpoints. It is simply a mechanism for user-defined function to be invoked at Open MPI's checkpoint, continue, and restart phases to support application-level C/R. Hence, the only data that is saved during the checkpoint is what is written in the application's checkpoint function - no MPI library state is saved. As such, the model for the self component is slightly different than, for example, the BLCR component. Specifically, the restart function is not invoked in the same process image of the process that was checkpointed. The restart phase is invoked during MPI_INIT of a new instance of the application (i.e., it starts over from main()).

This chapter presents an example application that takes advantage of the self CRS. Checkpointing and restarting of the MPI job occurs exactly as in the transparent cases (i.e., by using the `ompi-checkpoint` and `ompi-restart` tools or the API).

## 1. Interface

Open MPI uses `dlsym` to search for function signatures that can be used by the self CRS. This saves the application the inconvenience of being required to explicitly register these functions in the code.

```
// Default Checkpoint Callback
int opal_crs_self_user_checkpoint(char **restart_cmd);
// Default Continue Callback
int opal_crs_self_user_continue(void);
// Default Restart Callback
int opal_crs_self_user_restart(void);
```

If the application would rather explicitly register the function it may do so using the registration functions below.

```
// SELF CRS Checkpoint Registration Function
int OMPI_CR_self_register_checkpoint_callback(OMPI_CR_self_checkpoint_fn function);
// SELF CRS Callback Function Signature
typedef int (*OMPI_CR_self_checkpoint_fn)(char **restart_cmd);

// SELF CRS Continue Registration Function
int OMPI_CR_self_register_continue_callback(OMPI_CR_self_continue_fn function);
// SELF CRS Callback Function Signature
typedef int (*OMPI_CR_self_continue_fn)(void);

// SELF CRS Restart Registration Function
int OMPI_CR_self_register_restart_callback(OMPI_CR_self_restart_fn function);
// SELF CRS Callback Function Signature
typedef int (*OMPI_CR_self_restart_fn)(void);
```

### 2. Compiling

For the `dlsym` functionality to work the application must export symbols at compile time. Below is an example of exporting the symbols using `gcc` and the Open MPI wrapper compiler, `mpicc`.

```
shell$ mpicc my-app.c -export -export-dynamic -o my-app
```

### 3. Running

The application is launched as normal by specifying the ft-enable-cr AMCA parameter to `mpirun`. Optionally, the end user can specify the crs_self_prefix MCA parameter to help the `dlsym` function find function that are prefixed with the default names.

```
shell$ mpirun -np 2 -am ft-enable-cr my-app
shell$ mpirun -np 2 -am ft-enable-cr -mca crs_self_prefix my_personal my-app
```

### 4. Example Application

```
/*
 * Example Open MPI CRS 'self' program
 * Author: Josh Hursey
 */
#include <mpi.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>

#define LIMIT 100

/*** Function Declarations ***/
void signal_handler(int sig);

/* Default OPAL crs self callback functions */
int opal_crs_self_user_checkpoint(char **restart_cmd);
int opal_crs_self_user_continue(void);
int opal_crs_self_user_restart(void);
```

```
/*
 * OPAL CRS self callback functions. Use the following MCA parameter:
 * crs_self_prefix=my_personal
 */
int my_personal_checkpoint(char **restart_cmd);
int my_personal_continue(void);
int my_personal_restart(void);

/*** Global Variables ***/
int am_done = 1;
int current_step = 0;
char ckpt_file[128] = "my-personal-cr-file.ckpt";
char restart_path[128] = "/full/path/to/personal-cr";

/*** Main ***/
int main(int argc, char *argv[]) {
    int rank, size;
    current_step = 0;
    MPI_Init(&argc, &argv);

    /* So we can exit cleanly */
    signal(SIGINT, signal_handler);
    signal(SIGTERM, signal_handler);

    for(; current_step < LIMIT; current_step += 1) {
        printf("%d) Step %d\n", getpid(), current_step);
        sleep(1);
        if(0 == am_done) {
            break;
        }
    }
    MPI_Finalize();
    return 0;
}
void signal_handler(int sig) {
    printf("Received Signal %d\n", sig);
    am_done = 0;
}
```

```
/* OPAL crs self default callbacks for checkpoint */
int opal_crs_self_user_checkpoint(char **restart_cmd) {
    printf("opal_crs_self_user_checkpoint callback...\n");
    my_personal_checkpoint(restart_cmd);
    return 0;
}
int opal_crs_self_user_continue(void) {
    printf("opal_crs_self_user_continue callback...\n");
    my_personal_continue();
    return 0;
}
int opal_crs_self_user_restart(void) {
    printf("opal_crs_self_user_restart callback...\n");
    my_personal_restart();
    return 0;
}

/* OPAL crs self callback for checkpoint */
int my_personal_checkpoint(char **restart_cmd) {
    FILE *fp;
    *restart_cmd = NULL;
    printf("my_personal_checkpoint callback...\n");

    /* Open our checkpoint file */
    if( NULL == (fp = fopen(ckpt_file, "w")) ) {
        fprintf(stderr, "Error: Unable to open file (%s)\n", ckpt_file);
        return;
    }

    /* Save the process state */
    fprintf(fp, "%d\n", current_step);
    /* Close the checkpoint file */
    fclose(fp);
    /* Figure out the restart command */
    asprintf(restart_cmd, "%s", strdup(restart_path));

    return 0;
}
```

```
int my_personal_continue() {
    printf("my_personal_continue callback...\n");
    /* Don't need to do anything here since we are in the
     * state that we want to be in already. */
    return 0;
}

int my_personal_restart() {
    FILE *fp;

    printf("my_personal_restart callback...\n");

    /* Open our checkpoint file */
    if( NULL == (fp = fopen(ckpt_file, "r")) ) {
        fprintf(stderr, "Error: Unable to open file (%s)\n", ckpt_file);
        return;
    }

    /*
     * Access the process state that we saved and
     * update the current step variable.
     */
    fscanf(fp, "%d", &current_step);
    fclose(fp);

    printf("my_personal_restart: Restarting from step %d\n", current_step);
    return 0;
}
```

# D

# Nonblocking Process Creation and Management Operations

It is well established that many applications can see performance improvements by overlapping communication and computation. Nonblocking process management routines allow an application to overlap the creation of processes and/or establishment of communication channels with other computation. The amount of overlap can become substantial when creating or connecting a large number of processes. The proposal presented in this chapter focuses adding nonblocking interfaces to the existing interfaces described in Chapter 10 of the MPI 2.2 standard entitled “Process Creation and Management”.

The section titles in this chapter match those in the current standard to aid the reader in locating modifications to the original text.

## 1. Process Manager Interface

For all of the nonblocking routines described in this section an MPI_REQUEST object is returned. All of the completion calls (e.g., MPI_WAIT, MPI_TEST) are supported for these nonblocking routines. Upon returning from a completion call, the MPI_ERROR field of the MPI_STATUS object is set appropriately. The values of the MPI_SOURCE and MPI_TAG fields are undefined. The parameters marked as OUT should not be accessed until the request has been completed by one of the completion calls.

Matching blocking and nonblocking version of the MPI_ICOMM_SPAWN, MPI_ICOMM_SPAWN_MULTIPLE, MPI_ICOMM_ACCEPT, MPI_ICOMM_CONNECT, and MPI_ICOMM_JOIN operations is not allowed.

*Rationale*: The algorithms implemented for the blocking and nonblocking versions of these operations may not be equivalent for reasons of efficiency.

## 2. Starting Processes and Establishing Communication

| | |
|---|---|
| MPI_ICOMM_SPAWN(command, argv, maxprocs, info, root, comm, intercomm, array_of_errcodes, request) | |
| *IN* command | name of program to be spawned (string , significant only at root) |
| *IN* argv | arguments to command (array of strings , significant only at root) |
| *IN* maxprocs | maximum number of processes to start (integer, significant only at root) |
| *IN* info | a set of key−value pairs telling the runtime system where and how to start the processes (handle, significant only at root) |
| *IN* root | rank of process in which previous arguments are examined (integer) |
| *IN* comm | intracommunicator containing group of spawning processes (handle) |

| | |
|---|---|
| *OUT* intercomm | intercommunicator between original group and the newly spawned group (handle) |
| *OUT* array_of_errcodes | one code per process (array of integer) |
| *OUT* request | process creation request (handle) |

This call starts a nonblocking variant of MPI_COMM_SPAWN. It is erroneous to call MPI_REQUEST_FREE or MPI_CANCEL for the MPI_REQUEST associated with the MPI_ICOMM_SPAWN operation.

If a MPI_REQUEST for MPI_ICOMM_SPAWN or MPI_ICOMM_SPAWN_MULTIPLE is marked for cancellation using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the processes launched and communicator was created, or that the operation is canceled, in which case the processes are not launched and the communicator is not created. MPI_CANCEL can be called from any participating process.

*Advice to implementors*: The root can decide if it is able to or safe to cancel the request when it is notified of the cancellation request. If the processes have already started to be launched, the implementation is allowed to refuse the cancellation request if it is unable or unwilling to terminate the new processes. Alternatively, the implementation may decide that it is willing and able to terminate a newly launched process. Care should be taken when doing so since the process may cause side effects in the system. For example, if the launched process interacts with the file system before calling MPI_INIT this may influence already running processes.

*Side Note*: MPI_COMM_GET_PARENT does not have a nonblocking counterpart since it is a completely local operation.

**2.1. Starting Multiple Executables and Establishing Communication.** This call starts a nonblocking variant of MPI_COMM_SPAWN_MULTIPLE.

| MPI_ICOMM_SPAWN_MULTIPLE(count, array_of_commands, array_of_argv, array_of_maxprocs, array_of_info, root, comm, intercomm, array_of_errcodes) | |
|---|---|
| *IN* count | number of commands (positive integer, significant to MPI only at root) |
| *IN* array_of_commands | programs to be executed (array of strings, significant only at root) |
| *IN* array_of_argv | arguments for commands (array of array of strings, significant only at root) |
| *IN* array_of_maxprocs | maximum number of processes to start for each command (array of integer, significant only at root) |
| *IN* array_of_info | info objects telling the runtime system where and how to start processes (array of handles, significant only at root) |
| *IN* root | rank of process in which previous arguments are examined (integer) |
| *IN* comm | intracommunicator containing group of spawning processes (handle) |
| *OUT* intercomm | intercommunicator between original group and newly spawned group (handle) |
| *OUT* array_of_errcodes | one error code per process (array of integers) |
| *OUT* request | process creation request (handle) |

If a MPI_REQUEST for MPI_ICOMM_SPAWN or MPI_ICOMM_SPAWN_MULTIPLE is marked for cancellation using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the processes launched and communicator was created, or that the operation is canceled, in which case the processes are not launched and the communicator is not created. MPI_CANCEL can be called from any participating process.

**2.2. Nonblocking Spawn Example.** Below is an pseudo code example of a use of the nonblocking spawn interface.

```
int foo(int num_children) {
  for(i = 0; i < num_children; ++i) {
    MPI_Icomm_spawn("myapp", argv, 1, info, 0, MPI_COMM_SELF,
                            &intercomm[i], &array_of_statuses[i], &(requests[i]));
  }
  prepare_work_units();
  MPI_Waitall(num_children, requests, statuses);
  begin_work();
}
```

## 3. Establishing Communication

**3.1. Server Routines.** This call starts a nonblocking variant of MPI_OPEN_PORT. Since this might involve communication with an external name service, an application may want to perform computation while waiting on a response.

| MPI_IOPEN_PORT(info, port_name, request) | | |
|---|---|---|
| *IN* | info | implementation−specific information on how to establish an address (handle) |
| *OUT* | port_name | newly established port (string ) |
| *OUT* | request | connection request (handle) |

If a MPI_REQUEST for MPI_IOPEN_PORT is marked for cancellation by using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the port is opened, or that the operation is canceled, in which case the port is not opened.

| MPI_ICLOSE_PORT(port_name, request) | | |
|---|---|---|
| *IN* | port_name | a port (string ) |
| *OUT* | request | connection request (handle) |

This call starts a nonblocking variant of MPI_CLOSE_PORT. Again, since this might involve communication with an external name service computation overlap may be advantageous.

If a MPI_REQUEST for MPI_ICLOSE_PORT is marked for cancellation by using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the port is closed, or that the operation is canceled, in which case the port remains in its previous state.

| MPI_ICOMM_ACCEPT(port_name, info, root, comm, newcomm, request) | | |
|---|---|---|
| *IN* | port | port name (string , used only on root) |
| *IN* | info | implementation−specific information (handle, used only on root) |
| *IN* | root | rank in comm of the root node (integer) |
| *IN* | comm | intracommunicator over which call is collective (handle) |
| *OUT* | newcomm | intercommunicator with client as remote group (handle) |
| *OUT* | request | connection request (handle) |

This call starts a nonblocking variant of MPI_COMM_ACCEPT.

If a MPI_REQUEST for MPI_ICOMM_ACCEPT is marked for cancellation by using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the connection is established, or that the operation is canceled, in which case the any pending requests will be canceled returning MPI_ERR_NOT_CONNECTED to any processes waiting in MPI_COMM_CONNECT or MPI_ICOMM_CONNECT. MPI_CANCEL can be called from any participating process.

**3.2. Client Routines.** This call starts a nonblocking variant of the MPI_COMM_CONNECT operation.

| MPI_ICOMM_CONNECT(port_name, info, root, comm, newcomm, request) | | |
|---|---|---|
| *IN* | port_name | network address<br>(string , used only on root) |
| *IN* | info | implementation−specific information<br>(handle, used only on root) |
| *IN* | root | rank in comm of the root node<br>(integer) |
| *IN* | comm | intracommunicator over which call is collective<br>(handle) |
| *OUT* | newcomm | intercommunicator with client as remote group<br>(handle) |
| *OUT* | request | connection request<br>(handle) |

If a MPI_REQUEST for MPI_ICOMM_CONNECT is marked for cancellation by using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the connection is established, or that the operation is canceled, in which case the connection is not established. If the connection was establishing when the cancellation occurred then the matching MPI_COMM_ACCEPT or MPI_ICOMM_ACCEPT call may return MPI_ERR_NOT_CONNECTED. MPI_CANCEL can be called from any participating process.

**3.3. Name Publishing.** This call starts a nonblocking variant of the MPI_PUBLISH_NAME operation.

| MPI_IPUBLISH_NAME(service_name, info, port_name, request) | | |
|---|---|---|
| *IN* | service _name | a service name to associate with the port<br>(string ) |
| *IN* | info | implementation−specific information<br>(handle) |
| *IN* | port_name | a port name<br>(string ) |
| *OUT* | request | connection request<br>(handle) |

If a MPI_REQUEST for MPI_IPUBLISH_NAME is marked for cancellation by using MPI_CANCEL, then it must be the case that either the operation completed normally, in which

case the port is published, or that the operation is canceled, in which case the port remains unpublished.

| MPI_IUNPUBLISH_NAME(service_name, info, port_name, request) | | |
|---|---|---|
| *IN* | service_name | a service name (string) |
| *IN* | info | implementation−specific information (handle) |
| *IN* | port_name | a port name (string) |
| *OUT* | request | connection request (handle) |

This call starts a nonblocking variant of MPI_UNPUBLISH_NAME.

If a MPI_REQUEST for MPI_IUNPUBLISH_NAME is marked for cancellation by using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the port is unpublished, or that the operation is canceled, in which case the port remains in its previous state before the call to MPI_IUNPUBLISH_NAME.

| MPI_ILOOKUP_NAME(service_name, info, port_name, request) | | |
|---|---|---|
| *IN* | service_name | a service name (string) |
| *IN* | info | implementation−specific information (handle) |
| *OUT* | port_name | a port name (string) |
| *OUT* | request | connection request (handle) |

This call starts a nonblocking variant of MPI_LOOKUP_NAME.

If a MPI_REQUEST for MPI_ILOOKUP_NAME is marked for cancellation by using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the port_name variable contains the value found, or that the operation is canceled, in which case the port_name variable remains in its previous state before the call to MPI_ILOOKUP_NAME.

**3.4. Reserved Key Values.** The timeout key value is defined in an Advice to Implementors note in the MPI 2.0 standard notes for MPI_Comm_connect in Chapter 5, Section 5.4.3. We strengthen this commentary by standardizing a timeout for the MPI_INFO object that can be passed to either the blocking or non-blocking versions of both MPI_COMM_ACCEPT and MPI_COMM_CONNECT.

The timeout reserved key indicates the time out period for the MPI_COMM_ACCEPT, MPI_COMM_CONNECT, MPI_OPEN_PORT, MPI_PUBLISH_NAME, MPI_LOOKUP_NAME, and the MPI_UNPUBLISH_NAME operations. The value is specified as a positive integer representing the timeout in units of MPI_WTICK. The timeout is defined as the time it takes for the entire operation to complete. If the value is set to 0 then the timeout is set to an MPI implementation defined limit. The keyword is only meaningful at the root. If an implementation supports this reserved key it must behave as specified in this section. The standard does not specify how long an implementation should take to return to the application after a timeout. A good quality implementation will return to the application after a timeout within an implementation defined time bound

*Rationale*: The server might stall for a variety of reasons (e.g., fault recovery, overloaded). The client may want to limit their sensitivity to slow servers. The timeout key allows them to cancel a blocking connection establishment operation.

*Advice to implementors*: If the MPI implementation provides the timeout key, then it is suggested to set the default timeout to an indefinite timeout value. Additionally, the timeout for a collective operation should start once all processes arrive. This is important for the nonblocking variations of these connection establishment calls in order to account for process skew. Once the timeout has started, the timeout is between the roots representing the collective group in the paired operations. For example, the timeout starts on the server side once all processes have called MPI_COMM_ACCEPT or MPI_ICOMM_ACCEPT. The timeout starts on the client side once all processes have called MPI_COMM_CONNECT or MPI_ICOMM_CONNECT. The connection establishment operation is timed out between the root in MPI_COMM_ACCEPT and the root in MPI_COMM_CONNECT, not any other process in either intracommunicator.

MPI_COMM_CONNECT will try for the specified amount of time to establish a connection to the remote server. If the command cannot connect within the time specified, the library will raise either the error class MPI_ERR_NOT_RESPONDING or the error class MPI_ERR_NOT_AVAILABLE. The error class MPI_ERR_NOT_RESPONDING indicates that the remote server is alive, but has not responded to the connection request within the timeout bounds (i.e., no MPI_COMM_ACCEPT posted at the remote server). The MPI_ERR_NOT_AVAILABLE error class indicates that the remote server state is unknown, and has not responded to the connection request within the timeout bounds.

*Advice to implementors*: If the MPI implementation does not have the ability to distinguish between an active server and an unresponsive server, the implementation is allowed to return the stricter of the two error classes, namely MPI_ERR_NOT_AVAILABLE.

MPI_COMM_CONNECT will raise the error class MPI_ERR_NOT_CONNECTED if the connection to the remote server has started but could not be finished. This situation may be caused by a failure of the remote server during the connection establishment handshake. Additionally, this could be caused by a timeout on the operation when set by the application.

MPI_COMM_ACCEPT will try for the specified amount of time to accept a connection from a remote client. It is a valid behavior for the MPI_COMM_ACCEPT call to timeout with accepting connections, and should not be considered an error. This scenario is indicated by setting newcomm to MPI_COMM_NULL, and returning MPI_SUCCESS. MPI_COMM_ACCEPT will raise the error class MPI_ERR_NOT_CONNECTED if the connection to the remote client has started but could not be finished. This situation may be caused by a failure of the remote client during the connection establishment handshake.

If MPI_ERRORS_RETURN is not set on the communicator then the MPI_ERR_NOT_RESPONDING, MPI_COMM_NOT_AVAILABLE, and MPI_ERR_NOT_CONNECTED are fatal as defined by the default of MPI_ERRORS_ARE_FATAL, even though the error code may have been caused by the timeout on the operation.

*Rationale*: It was recognized that the application, by setting the timeout key value, may expect that the errors caused by the timeout of the operation should not be fatal. However, by making this class of errors non-fatal, will cause previous standard conformant

applications to break since functions that they assumed would only return MPI_SUCCESS would then return a wider class of errors.

**3.5. Releasing Connections.** This call starts a nonblocking variant of MPI_COMM_DISCONNECT.

| MPI_ICOMM_DISCONNECT(comm, request) | | |
|---|---|---|
| *INOUT* | comm | communicator (handle) |
| *OUT* | request | connection request (handle) |

If a MPI_REQUEST for MPI_ICOMM_DISCONNECT is marked for cancellation using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the communicator is disconnected, or that the operation is canceled, in which case the communicator is still valid.

**3.6. Another Way to Establish MPI Communication.** This call starts a nonblocking variant of MPI_COMM_JOIN.

| MPI_ICOMM_JOIN(fd, intercomm, request) | | |
|---|---|---|
| *IN* | fd | socket file descriptor |
| *OUT* | intercomm | new intercommunicator (handle) |
| *OUT* | request | connection request (handle) |

If a MPI_REQUEST for MPI_ICOMM_JOIN is marked for cancellation by using MPI_CANCEL, then it must be the case that either the operation completed normally, in which case the communicator is created, or that the operation is canceled, in which case the communicator is not created.

The file descriptor must not be used between the call to MPI_ICOMM_JOIN and corresponding call to the request completion call. Upon return from request completion call on the request from MPI_ICOMM_JOIN, the file descriptor will be open and quiescent.

# Joshua J. Hursey

## Contact Information

Computer Science Department
Lindley Hall 215
150 S. Woodlawn Ave.
Bloomington, IN 47405-7104
Email: jjhursey@cs.indiana.edu
Web: http://www.cs.indiana.edu/~jjhursey

## Research Interests

My research interests span parallel and distributed systems, scientific computing, and software engineering. I am currently focusing on the development of scalable fault tolerance and resiliency techniques for High Performance Computing applications.

## Education

| | |
|---|---|
| July 2010 | Ph.D., Computer Science<br>Indiana University, Bloomington, IN |
| May 2006 | M.S., Computer Science<br>Indiana University, Bloomington, IN |
| May 2003 | B.A., Computer Science, Mathematics Minor<br>Earlham College, Richmond, IN |

## Publications

### Journal

- Alex Breuer, **Joshua J. Hursey**, Tonya Stroman, and Arvind Verma. Visualization of criminal activity in an urban population. *Artificial Crime Analysis Systems: Using Computer Simulations and Geographic Information Systems*, January 2008.

### Conference and Workshop

- **Joshua Hursey**, Chris January, Mark O'Connor, Paul H. Hargrove, David Lecomber, Jeffrey M. Squyres, and Andrew Lumsdaine. Checkpoint/restart-enabled parallel debugging. *Proceedings of the European MPI Users Group Conference (EuroMPI)*, September 2010.
- **Joshua Hursey**, Timothy I. Mattox, and Andrew Lumsdaine. Interconnect agnostic checkpoint/restart in Open MPI. In *Proceedings of the 18th ACM International Symposium on High Performance Distributed Computing (HPDC 2009)*, June 2009.
- Joseph A. Cottam, Andrew Lumsdaine, and **Joshua Hursey**. Representing unit test data for large scale software development. In *ACM Sypmposium on Software Visualization (SoftVis 2008)*, September 2008.
- **Joshua Hursey**, Ethan Mallove, Jeffrey M. Squyres, and Andrew Lumsdaine. An extensible framework for distributed testing of MPI implementations. In *Proceedings of the 14th European PVM/MPI Conference*, October 2007.

- Yvonne Rogers, Kay Connelly, Lenore Tedesco, William Hazlewood, Andrew Kurtz, Robert E. Hall, **Joshua Hursey**, and Tammy Toscos. Why it's worth the hassle: The value of in-situ studies when designing Ubicomp. In *UbiComp 2007: Ubiquitous Computing*, September 2007. Nominated for the Best Paper Award.

- **Joshua Hursey**, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, March 2007.

- David S. Wise, Craig Citro, **Joshua Hursey**, Fang Liu, and Michael Rainey. A paradigm for parallel matrix algorithms: Scalable Cholesky. In *Proceedings of the 11th International Euro-Par Conference*, August 2005.

**Technical Reports**

- **Joshua Hursey**, Jeffrey M. Squyres, and Andrew Lumsdaine. A checkpoint and restart service specification for Open MPI. Technical Report TR635, Indiana University, Bloomington, Indiana, USA, July 2006.

- Craig Shue, **Joshua Hursey**, and Arun Chauhan. MPI over scripting languages: Usability and performance tradeoffs. Technical Report TR631, Indiana University, Bloomington, Indiana, USA, February 2006.

**Posters**

- **Joshua Hursey**, Scott Hampton, Pratul Agarwal, and Andrew Lumsdaine. An adaptive checkpoint/restart library for large scale HPC applications. In *SIAM Conference on Parallel Processing and Scientific Computing (PP10)*, February 2010.

- Ralph Castain, **Joshua Hursey**, Timothy I. Mattox, Chase Cotton, Robert M. Broberg, and Jonathan M. Smith. A resilient runtime environment for HPC and internet core router systems. In *IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC09)*, November 2009.

- Joseph A. Cottam, **Joshua Hursey**, and Andrew Lumsdaine. SeeTest: Unit test visualization. In *IEEE Symposium on Information Visualization (InfoVis 2008)*, November 2008.

- Joseph A. Cottam, **Joshua Hursey**, and Andrew Lumsdaine. SeeTest: Unit test visualization. Indiana University Computer Science and Informatics Graduate Research Poster Session, May 2008. Recieved honorable mention award.

- **Joshua Hursey**, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. Indiana University Computer Science and Informatics Graduate Research Poster Session, March 2007.

- Charles Peck, **Joshua Hursey**, Joshua McCoy, and John Schaefer. Calculating 1/sqrt(x) for molecular dynamics packages on commodity vector architectures. SIAM Conference on Computational Science and Engineering, February 2005.

- Charles Peck, **Joshua Hursey**, Joshua McCoy, and John Schaefer. Folding@Clusters: Harnessing grid-based parallel computing resources for molecular dynamics simulations. SIAM Conference on Computational Science and Engineering, February 2005.

- Charles Peck, **Joshua Hursey**, and Joshua McCoy. Benchmarking and tuning the GROMACS molecular dynamics package on beowulf clusters. SIAM Conference on Parallel Processing for Scientific Computing, February 2004. Received Best Poster Award.

- Charles Peck, **Joshua Hursey**, and Joshua McCoy. Benchmarking and tuning the GROMACS molecular dynamics package on beowulf clusters. Earlham College Undergraduate Research Conference, November 2003.
- **Joshua Hursey**. Fingerprint minutiae classification: A multi-layer feed-forward artificial neural network approach. Butler University's 14th Annual Undergraduate Research Conference, April 2003.

**Work Under Submission**

- **Joshua Hursey** and Andrew Lumsdaine. A composable runtime recovery policy framework supporting resilient HPC applications. Under submission, 2010.

**Invited Talks**

- **Joshua Hursey**, Jeffrey M. Squyres, Abhishek Kulkarni, and Andrew Lumsdaine. Open MPI tutorial. Indiana University Booth at IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC09), November 2009.
- **Joshua Hursey**. A transparent process migration framework for Open MPI. Cisco Systems, Inc. Booth at IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC09), November 2009.
- **Joshua Hursey**. Fault tolerance in Open MPI. Los Alamos National Laboratory Resilience Seminar, November 2009.
- **Joshua Hursey**. Fault tolerance in High Performance Computing: MPI and checkpoint/restart. Indiana University Booth at IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC08), November 2008.
- **Joshua Hursey**. MPI implementation health assessment through multi-institutional distributed testing. Cisco Systems, Inc. Booth at IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC08), November 2008.
- **Joshua Hursey**. Checkpoint/restart support in Open MPI. Sun Microsystems, Inc. Technical Talk Series, May 2008.
- **Joshua Hursey**. Process fault tolerance and Open MPI. Oak Ridge National Laboratory, June 2007.
- **Joshua Hursey**. Process fault tolerance in Open MPI. Innovative Computing Laboratory (ICL) Friday Lunch Speaker Series, University of Tennessee, Knoxville, February 2007.
- **Joshua Hursey**. Dealing with disaster: Fault tolerance in Open MPI. Indiana University Booth at IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC06), November 2006.
- **Joshua Hursey**. Fault tolerance and MPI. Los Alamos National Laboratory CCS1 Brown Bag Speaker Series, July 2006.

## Selected Honors and Awards

- Honorable Mention, Computer Science and Informatics Graduate Research Symposium, Indiana University, 2008.
- Nominated for Best Paper Award, UbiComp 2007: Ubiquitous Computing, 2007.
- Best Poster Award, SIAM Conference on Parallel Processing for Scientific Computing, 2004.
- Academic and Departmental Honors Degree, Earlham College, 2003.

## Research Experience

Jan. 2005 - Present **Research Assistant** Indiana University
Advisor: Professor Andrew Lumsdaine Bloomington, IN

Conducted research into scalable process fault tolerance techniques for High Performance Computing (HPC). Developed a transparent, coordinated checkpoint/restart technique for the Open MPI project. Assisted in integrating support for the CIFTS FTB project into Open MPI.

May 2009 - Aug. 2009 **Research Assistant** Oak Ridge National Labs.
Advisor: Dr. Richard Graham Oak Ridge, TN

Conducted research into scalable, resilient runtime environments for HPC. Contributed to the Open MPI and Scalable Tools Communication Infrastructure (STCI) projects.

June 2007 - Aug. 2007 **Research Assistant** Oak Ridge National Labs.
Advisor: Dr. Richard Graham Oak Ridge, TN

Integrated Cray Application Level Placement Scheduler (ALPS) support into the Open MPI project to support the Cray XT4/5 (Jaguar). Supported Open MPI development and deployment at ORNL. Contributed to the MPI Testing Tool (MTT) to support Cray environments, back-end database enhancements, and user interface improvements.

May 2006 - Aug. 2006 **Research Assistant** Los Alamos National Labs.
Advisor: Dr. David Daniel Los Alamos, NM

Conducted research into resilient runtime environments, and checkpoint/restart techniques for HPC. Designed and developed transparent, coordinated checkpoint/restart process fault tolerance for the Open MPI project.

May 2005 - Aug. 2005 **Research Assistant** Lawrence Berkeley National Labs.
Advisor: Dr. Paul H. Hargrove Berkeley, CA

Supported efforts to make the K42 operating system suitable for HPC. Conducted research into checkpoint/restart process fault tolerance for the Open MPI project.

Aug. 2004 - Dec. 2004 **Research Assistant** Indiana University
Advisor: Professor David Wise Bloomington, IN

Conducted research into cache oblivious matrix multiplication algorithms on HPC clusters.

Aug. 2003 - Aug. 2004 **Research Assistant** Earlham College
Advisor: Professor Charles Peck Richmond, IN

Designing and developing the Folding@Clusters distributed computing project. Developed methodologies for benchmarking and tuning interdependent scientific software packages on Beowulf clusters.

Nov. 2002 - May 2005 **Software Consultant** Safe Passage Comm., Inc.
Manager: Charles Peck Richmond, IN

Designing and developing mobility applications for PDAs, cellular phones, Symbol scanners, and other wireless devices. Contributed to open source projects supporting these applications.

June 2002 - Aug. 2003 **Research Assistant** Earlham College
Advisor: Professor Charles Peck Richmond, IN

Conducted research into software design and structure of molecular dynamics applications for Beowulf clusters.

Aug. 2002 - Dec. 2002 **Senior Seminar Project** Earlham College
Advisor: Professor Jim Rogers Richmond, IN

Conducted research into fingerprint biometric recognition using artificial neural networks..

## Teaching Experience

Sept. 2008 - Dec. 2008 **Primary Instructor** Indiana University
Introduction to Operating Systems Bloomington, IN
(ACM/IEEE-CS CS222w)

Text: Silberschatz, A., Galvin, P.B., Gagne, G. *Operating Systems Concepts*, Eighth Edition, John Wiley & Sons, Inc., 2008.

Jan. 2004 - June 2004 **Adjunct Instructor** Purdue University
Introduction to C++ Programming Richmond, IN
(ACM/IEEE-CS CS101O)

Text: Lafore, R. *Object-Oriented Programming in C++*, Forth Edition, Sams, 2002.

Aug. 2003 - Dec. 2003 **Adjunct Instructor** Earlham College
Programming and Problem Solving Richmond, IN
(ACM/IEEE-CS CS101I)

Text: Savitch, W. *Problem Solving with C++: The object of programming*, Forth Edition, Addison Wesley, 2003.

May 2003 - June 2003 **Teaching Assistant, Co-Organizer** Earlham College
Service Learning Trip: Jamaica Richmond, IN

Service learning trip to Jamaica with 15 students and faculty members.

## Service

**Professional**

- **Program Committee Member:**
  EuroMPI Users' Group Conference, 2010.
- **Reviewer:**
  IEEE Transactions on Parallel and Distributed Systems, 2008;
  IEEE International Conference on e-Science and Grid Computing, 2007;
  Journal of Parallel and Distributed Computing, 2007;
  Concurrency and Computation: Practice and Experience, 2007;
  IEEE International Symposium on Cluster Computing and the Grid (CCGrid), 2007.

**Computer Science Department**

- **Moderator**, Academic Survival Graduate Student Colloquia Series (ASGSC) (a series of research and professional development events for graduate students including a Faculty Lightning Talks event involving 18 faculty members), 2008-2009.
- **Coordinator**, Kill-the-Week Social Hour, Computer Science Graduate Student Association, 2005 - 2007.

# Affiliations

- Association for Computing Machinery (ACM) (2003 - Present)
- Institute of Electrical and Electronics Engineers (IEEE) (2008 - Present)
- Society for Industrial and Applied Mathematics (SIAM) (2009 - Present)

# References

Available upon request.