Toward a Transparent, Checkpointable Fault-Tolerant Message Passing Interface for HPC Systems

by

Nawrin Sultana

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
December 14, 2019

Keywords: MPI, Fault-Tolerant, Checkpoint, Exascale

Approved by

James Cross, Chair, Professor of Computer Science and Software Engineering
Anthony Skjellum, Co-chair, Professor of Computer Science, University of Tennessee at Chattanooga
David Umphress, Professor of Computer Science and Software Engineering
Xiao Qin, Professor of Computer Science and Software Engineering
Purushotham Bangalore, Professor of Computer Science, University of Alabama at Birmingham

Abstract

With each successive generation of large-scale high-performance computing (HPC) systems, faults and associated failures are becoming more frequent. Long-running applications in such systems require efficient fault-tolerance support. The Message Passing Interface (MPI) is the de facto standard for HPC message passing middleware since its first release in 1994. However, the MPI Standard itself does not provide any mechanism to continue running MPI after a failure. Thus, there is a growing effort in the MPI community to incorporate fault-tolerance constructs into MPI. When an MPI program experiences a failure, the most common recovery approach is to restart all processes from a previous checkpoint and to re-queue the entire job which incurs unnecessary overhead. The purpose of this dissertation is to design a fault-tolerant MPI for Bulk Synchronous Parallel (BSP) applications with the goals of efficient failure recovery as well as easy adoption in large-scale production applications. This dissertation describes a new fault-tolerant model for MPI called “MPI Stages”. We discuss the design, applicability, and performance of MPI Stages recovery model. Additionally, we provide the minimal MPI semantics for applications and libraries to use MPI Stages. To demonstrate this new model, we introduce “ExaMPI”, a modern C++ implementation of a subset of MPI-3.x functionality. In addition, we analyze applications that use MPI programming model to understand the most commonly used features and characteristics of MPI in next-generation exascale systems.

Acknowledgments

I would like to express my sincere gratitude to my co-advisor (former advisor) Professor Anthony Skjellum for his supervision, inspiration, and overwhelming support during my doctoral study. I'm thankful for all the guidance and opportunities that he has provided me throughout my Ph.D. study for my academic and professional development.

I would also like to thank my committee chair Professor James Cross and the committee members— Professor David Umphress, Professor Xiao Qin, Professor Purushotham Bangalore for their valuable insights and encouragement in completion of my degree. A special thanks to Professor Robert Nelms for being the external reader of my dissertation.

Furthermore, I express my sincere gratitude to my collaborators Ignacio Laguna and Kathryn Mohror from Lawrence Livermore National Laboratory (LLNL) for mentoring and providing me with valuable technical knowledge throughout this work.

I would specifically like to thank my fellow colleague Martin Rüfenacht for his many contributions and support in completion of my work. I would also like to thank Derek Schafer and Shane Farmer for their contributions to my work.

Also, I acknowledge the support by the department of Computer Science and Software Engineering at Auburn University.

Finally, to my spouse, Animesh Mondal and my parents, I cannot thank you enough for your continued support.

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 and with partial support from the National Science Foundation under Grants #'s 1562659, 1562306, 1617690, 1822191, 1821431, 1918987.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
BLCR	Berkeley Lab Checkpoint/Restart
BSP	Bulk Synchronous Parallel
CPR	Checkpoint/restart
ECP	Exascale Computing Project
FA-MPI	Fault-Aware MPI
FTI	Fault-Tolerance Interface
FT-MPI	Fault-Tolerant MPI
HPC	High Performance Computing
I/O	Input/Output
LFLR	Local Failure Local Recovery
MM	Matrix Multiplication
MPI	Message Passing Interface
MTBF	Mean-Time between Failure
P2P	Point-to-Point
PMPI	Profiling Message Passing Interface

RMA	Remote Memory Access
RTS	Run-Through Stabilization
SCR	Scalable Checkpoint/restart
SMP	Symmetric Multi-processor
SSD	Solid State Drive
ULFM	User-Level Failure Mitigation
UML	Unified Modeling Language

Chapter 1

Introduction

1.1 Motivation

The computation power of high-performance computing (HPC) systems is expected to increase from Peta (10^{15}) to Exa (10^{18}) Floating Point Operations (FLOPs). The coming exascale systems will require the simultaneous use and control of millions of processing, storage, and networking elements. Based on data from top500.org, the peak performance of the top supercomputers doubles roughly every 14 months [1]. To maintain this performance growth prediction, system designers should increase the processing power through a combination of increase in microprocessor clock frequency, the number of sockets, and the number of cores. Because of power limitations of transistors, microprocessor clock frequency remains relatively constant nowadays [2]. So, to increase performance, exascale systems require to increase the number of components. An exascale system is expected to have thousands of cores per computing node, with millions computing cores to reach the required level of performance. The first two exascale supercomputers in US, Aurora and Frontier will be built by 2021. Table 1.1 shows the performance of top two petascale and exascale (expected) supercomputers in US.

As the hardware components and the software complexities increase in HPC systems, the failure rate is also anticipated to increase. One of the hardest problems in these systems will be the ability to provide reliability and availability. With a large number of hardware components involved, future exascale systems are expected to exhibit much higher fault rates than current systems. Such increase in the failure rate will decrease the Mean-Time Between Failures (MTBF) of the system. It is anticipated that the mean-time to failure could be less than

Table 1.1: Performance of Petascale and Exascale Systems

Rank	Petascale			Exascale (expected)		
	System	Performance (RPEAK)	Cores	System	Performance (RPEAK)	Cores
1	Summit	200 PFLOPS	2,414,592	Frontier	∼1.5 EFLOPS	N/A
2	Sierra	125 PFLOPS	1,572,480	Aurora	∼1 EFLOPS	N/A

one hour in exascale systems [3, 4]. This frequent failure will cause difficulty for applications to make forward progress. Long-running scientific applications that are expected to run on future exascale systems will require better application-level support to tolerate frequent failures efficiently [5].

The Message Passing Interface (MPI) is the de facto standard for HPC message passing middleware. Most HPC applications use MPI data communication. The HPC community is likely to continue using MPI in future exascale systems for its flexibility, portability, and efficiency [6]. Although MPI is a widely used programming model for large-scale scientific applications, it has limited support for fault tolerance. The MPI Standard itself does not provide any mechanism to continue running MPI after a failure. The MPI Standard states:

“After an error is detected, the state of MPI is undefined.” (MPI Standard 3.1, Page 340)

However, MPI provides several predefined error handlers. Also, an application can use its own error handler. The default behavior of MPI is to abort all running processes if case of a process failure. To overcome this shortcoming, several approaches have been proposed to address how failures can be tolerated in MPI. Although it is agreed that better fault tolerance solutions for MPI would be beneficial, the question of how and at what level these must be implemented has yet to be answered. Existing fault-tolerance programming models for MPI applications can be broadly classified as (i) rollback recovery: restart MPI application from a previously saved program state; (ii) error code dependent: checking returned error codes per MPI operation where, if an error is detected, recovery takes place; (iii) try/catch blocks, where groups of MPI operations are protected; (iv) global-restart, where the state of MPI is globally reinitialized and cleaned up upon a failure. None of these has yet been integrated to the MPI Standard. Many of them are too difficult to use or too complex to adopt in existing codes. As a result, in HPC

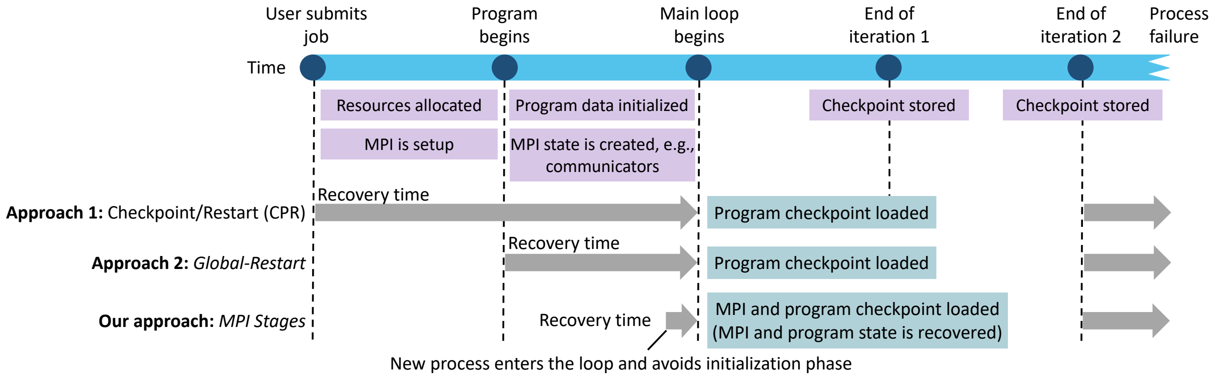

User submits job
Program begins
Main loop begins
End of iteration 1
End of iteration 2
Process failure
Time
Resources allocated
MPI is setup
Program data initialized
MPI state is created, e.g., communicators
Checkpoint stored
Checkpoint stored
Recovery time
Approach 1: Checkpoint/Restart (CPR)
Program checkpoint loaded
Recovery time
Approach 2: Global-Restart
Program checkpoint loaded
Our approach: MPI Stages
Recovery time
MPI and program checkpoint loaded (MPI and program state is recovered)
New process enters the loop and avoids initialization phase

Figure 1.1: Overview of recovery time (grey arrows) of a process failure for different fault-tolerance approaches.

applications, rollback recovery or checkpoint/restart (CPR) is still the most common approach of fault tolerance. So, any new fault-tolerant method needs to be easily adoptable with existing codes. Most HPC applications follow bulk synchronous programming (BSP) model. Based on the programming complexity, feasibility of implementation and performance, the global-restart model is the best option for large-scale, bulk synchronous codes as it resembles the fault tolerance semantics of CPR. The ideal solution for developers of MPI-based applications is that the system (or the runtime) transparently replace a failed process, which implies that it has to restore the previous state prior to the failure. While one can retrieve and restore the application data, one cannot do so with MPI state. This is why we need a solution that allows to save (resp, retrieve) the MPI state before (resp, after) failures.

In this dissertation, we present a new global-restart approach `MPI Stages` for transparent fault tolerance support to MPI. This approach requires minimal application code refactoring; hence it is easy to adopt in large-scale HPC codes. In our approach, the failed processes are transparently replaced by the MPI runtime and the state of MPI and application are recovered from a saved checkpoint.

Motivating Example. Bulk synchronous HPC applications typically follow a common programming pattern that comprises two main components: *program initialization* and a *computation loop*. At the beginning of the program, the program initializes its data, then calls `MPI_Init()` to create MPI state. Then the program enters the main computation loop and every n iteration in this loop, the program takes a checkpoint of its state.

Let us consider a bulk synchronous application executing the main computation in a loop. Here, we assume that a process has failed during iteration 3. Figure 1.1 shows the failure recovery time using different fault tolerance approaches. For traditional CPR (approach 1), a process failure triggers an entire job restart. The recovery starts by the user resubmitting the job. The system allocates all the resources again for the MPI job. The program initializes all data and `MPI_Init` creates all MPI state. Then, the program loads the application checkpoint and starts execution from iteration 3. Thus, recovery time includes a new job submission, delay while the program is selected to run in the queue again, program and MPI state initialization, and loading the program checkpoint.

Approach 2 in Figure 1.1, global-restart, reduces recovery time since all processes start from the beginning of the program without a job restart. Global-restart then initializes all program data and reinitializes MPI. Then, the execution starts from iteration 3 by loading the program checkpoint. As shown in Figure 1.1, the recovery time includes program and MPI state reinitialization and loading the program checkpoint, the sum of which is faster than the traditional CPR. Because the job is not restarted, this method avoids job/resource re-allocation time, enabling the reuse of existing process connections, and enabling the application to load checkpoints from memory (if they exist) for live processes.

Our Approach: MPI Stages. In our approach, the goal is to allow all processes to start execution from the main computation loop instead of from the main program. We introduce the concept of an MPI state checkpoint to replace a failed process transparently [7, 8, 9, 10]. At the end of n iterations (as defined by the application), we store a checkpoint of MPI state along with the program state. Figure 1.1 shows that, all processes start from iteration 3 by loading both the program and MPI state, which removes the requirement of program data and MPI state initialization for the live processes along with the need for job restart.

In global-restart, all processes perform the reinitialization in parallel. The initialization phase is application dependent so the recovery time will vary based on it.

In MPI Stages, recovery time is independent of the application initialization time since live processes avoid it.

At high-level, the benefits from our method are:

- **Reduced Recovery Time:** We replace the failed process with a new instance that repairs both the program and MPI state faster than previous approaches as no initialization of state is required. This leads to faster recovery as the new process can quickly continue from where the failed one stopped.

- **Ease of Adoption:** As shown in Figure 1.1, traditional CPR takes a program checkpoint after a certain number of iterations. Our approach matches the semantics of CPR. It adds an extra checkpoint for MPI state along with that program checkpoint; hence, it is easy to adopt.

- **Complete Recovery:** The program does not shrink the communicator(s) involved as in other approaches (e.g., ULFM), thus allowing the application to continue without complex reconfigurations (e.g., continuing execution with $n - 1$ processes).

- **No Language Dependency:** We do not use any programming language specific semantics. The Reinit model uses setjmp/longjmp of C language which is not supported in Fortran. Our model is applicable in both C and Fortran MPI program.

A fundamental challenge with this model is that we must be able to manage and capture MPI state during normal operation in a meaningful and complete way. Our goal is to develop a prototype of `MPI Stages` as a proof-of-concept. However, the difficulty of deep experimentation with existing MPI implementations—which are quite large and complex—substantially raises the cost and complexity of proof-of-concept activities. It also limits the community of potential contributors to new and better MPI features and implementations alike. Also, we find that most MPI applications use a small set of functionality from MPI Standard [11, 12, 13], which means the complexity associated with full API support isn't needed for many kinds of applications and, hence, application experiments. These motivate us to introduce ExaMPI [14], a modern C++ implementation of a subset of MPI-3.x. The goal is to enable researchers to experiment rapidly and easily with new concepts, algorithms, and internal protocols for MPI. We implement a prototype of our fault-tolerant model on ExaMPI.

1.2 Dissertation Statement

Scientific applications that are expected to run on future exascale systems will require better application-level support to tolerate frequent failures efficiently. The purpose of this dissertation is to demonstrate an efficient approach to fault tolerance for HPC applications that use MPI programming model. Our goal is to design an application-level fault-tolerant MPI that will achieve faster recovery from failure compared to the state-of-the-practice models as well as easily conform with existing codebases with no language dependency. This dissertation achieves its goal as follows:

- Characterize the communication and usage patterns of MPI in exascale applications.
- Design and implement a new research MPI library with a subset of functionality to focus on fault tolerance and optimization on such subsets.
- Design and prototype a checkpoint-based fault-tolerant model for MPI that transparently replaces the failed processes.
- Demonstrate the performance of our recovery model by comparing it with other fault-tolerant models of MPI.

1.3 Audience

The audience of this dissertation are both MPI and application developers and researchers who seek to investigate new and diverse functionality with MPI.

- HPC developers and researchers investigating in fault-tolerance for large-scale systems may find MPI Stages very useful. Also, the Fault-Tolerant Working Group (FTWG) in MPI Forum may utilize some features of MPI Stages in the standardization of MPI in presence of failure.
- MPI implementation developers and HPC centers can utilize the pattern of MPI usage in exascale applications to allocate resources for optimizing the most commonly used MPI features.

- MPI researchers can utilize our experimental MPI library for quick and easy prototype implementation of new features and ideas.

1.4 Outline

The remainder of this dissertation is organized as follows: Chapter 2 provides background work and related literature about fault-tolerance in parallel systems. Chapter 3 describes the characterization of future exascale applications using static and dynamic analysis. Chapter 4 introduces the design and implementation of the new MPI implementation, `ExaMPI`. Chapter 5 introduces the design and implementation of the failure recovery model, `MPI Stages` along with MPI API extension required to use it in applications and libraries. Chapter 6 presents the experimental design and evaluation of `MPI Stages`. Chapter 7 discusses the future work related to this dissertation. Finally, we summarize the research in Chapter 8.

Chapter 2

Background and Related Work

There are several methods available to tolerate failure in MPI. However, the increase of machine scale necessitates the exploration of new ideas of fault tolerance as the lack of appropriate resilience solutions is expected to be a major problem at exascale. In this chapter, we provide background information and related literature on resilience to establish adequate context for the rest of the dissertation.

2.1 Terminology

In this section, we offer the basic terminologies related to resilience in parallel and distributed systems.

Error— An error is the deviation of one or more external states of the system from the correct service state.

Fault— The hypothesized cause of an error is called a fault. Only when a fault becomes active it results in an *error*, otherwise it is dormant. Faults can occur in both development and operational phases of a system. Here, we only address the operational faults. Software, hardware, environment, and human can cause faults. However, software and hardware faults are predominant in supercomputers with hardware responsible for more than 50% and software around 20% of all failures [2].

Failure— Failure is the diversion of a system from its correct behavior. One or more errors can become effective and cause a system to encounter failure. A failure in a subsystem or a component can cause a fault in other parts of the system. Failure can happen at any part of software stack, operating system and kernel. The cause of software errors can be bugs in the implementation. File system and disk failures happen at the storage level. Even planned or unplanned hardware maintenance can cause applications to fail.

Fault Tolerance— The ability of a system to survive failures. There are several phases of fault tolerance (e.g., error detection, notification, recovery). Different levels of the stack (e.g., hardware, system, application software) can be involved in fault tolerance of a system.

Resilience— The ability to keep applications executing to a correct result despite underlying system faults. We use it interchangeably with fault tolerance in this dissertation.

Mean-Time To Failure (MTTF)— The average time it takes for a system to fail after recovering from the last failure.

Mean-Time Between Failures (MTBF)— The average time between two consecutive failures in a system.

Fail-stop failure— A condition in which a running process stops responding because of reasons such as: node failure caused by defective hardware, process crash caused by segmentation faults and other software related issues. It impacts one or more processes and needs application intervention for recovery.

Transient soft faults— Transient soft fault happens in memory through unwanted bit-flips. Causes of these faults can be cosmic rays, low voltage, or simply the age of components. Some transient failures are detectable while others not.

Byzantine fault— When the sequence of steps of a system become arbitrary or malicious, hence, any type of behavior can be observed. Byzantine fault can lead a system from one valid state to another valid, but incorrect state.

2.2 Large-Scale Programming Models

In this section we provide background on several popular parallel programming models. We detail different aspects of Message Passing Interface (MPI) programming model, which will help us explain some of our findings later on.

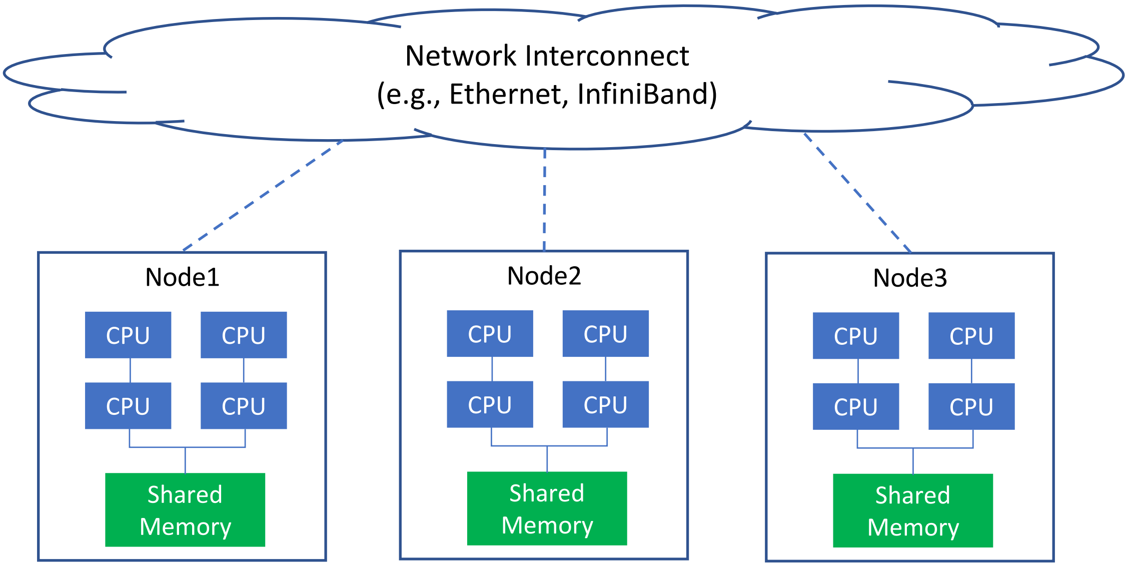


Figure 2.1: Overview of SMP architecture in cluster systems.

2.2.1 Message Passing Interface

MPI is a message passing library standard based on the consensus of the MPI Forum, which involve approximately 40 organizations that include major vendors of concurrent computers, researchers from universities, government laboratories, and industry mainly from United States and Europe. The goal of MPI is to establish a portable, efficient, and flexible standard for writing message passing programs. Because of its portability, efficiency, and flexibility, MPI has become the 'standard' for writing message passing programs on HPC platforms.

MPI primarily addresses the message-passing parallel programming model: moves data from the address space of one process to another process through cooperative operations on each process. Originally, MPI was designed for distributed memory architectures. However, as architecture trends changed, shared memory Symmetric Multi-processors (SMPs) were combined over networks creating hybrid distributed/shared memory systems. Figure 2.1 shows the SMP architecture used in cluster system. MPI libraries adapted itself to handle both distributed and shared memory SMP architectures. It also supports different network interconnects (e.g., Sockets, InfiniBand).

MPI Standard. The MPI standardization effort was formally started in early 1993 to facilitate the development of parallel applications and libraries. The first MPI specification (version

1.0) was released in May 1994. Since then the MPI Forum has released several versions (major and minor) of the MPI Standard. Several new features (e.g., dynamic process management, one-sided communication, MPI I/O) have been introduced in version 2.0. Another major update to the MPI Standard is version 3.0. The MPI-3.0 standard contains significant extensions to MPI functionality, including non-blocking collectives, new one-sided communication operations, and Fortran 2008 bindings. The latest version of MPI Standard is MPI-3.1 [15]. The MPI Forum is currently working on MPI-4.0.

MPI Implementation. Although the MPI programming interface has been standardized, actual library implementations differ from each other. The implementations vary based on several factors, such as the supported version of MPI Standard, supported features, supported network interfaces, how are MPI jobs launched, etc.

MPI implementations have existed since 1993, commencing with MPICH [16]. Over the past 26 years, MPICH, Open MPI [17], LAM/MPI [18], and other open source MPIs have grown in size, complexity, support, and usership. Commercial MPI products based on original code (open source derivatives) were also created, and some of them are still in use today.

During this time, there has been consistent and even growing interest in experimenting with MPI in terms of additions, changes, and enhancements to implementations and functionality. Commercial and free derivative products of these open implementations have also been successful, such as Cray MPI [19], Intel MPI [20], IBM Spectrum MPI [21], and MVAPICH [22].

2.2.2 Bulk Synchronous Parallel

The Bulk-Synchronous Parallel (BSP) model was proposed by Valiant as a standard interface between parallel software and hardware [23]. The BSP model consist of three parts: the component which performs memory operation and computation jobs, the router which works as a message transfer unit between components, and the barrier synchronization unit for synchronizing all memory and data transfer units. Basically, in the BSP model, a component calls a number of local computations and remote data transfer operations in a time limit L as a superstep (or just number of operations instead of time for measuring limit), then all operations are synchronized using a barrier (simply blocks until all operations are completed). Most of todays scalable applications in large government supercomputers are based on the BSP model.

2.2.3 Charm++

Charm++ [24] was introduced as a language similar to C++ with a few extensions such as movable objects or tasks called Chares. Chares can be moved dynamically in the system and be placed in appropriate locations and provide a better support for load balancing and performance tuning. Failed chares can be moved to a safe location without restarting the entire application stack. Charm++ separates sequential and parallel objects. One of current Charm++ implementations is on top of MPI.

2.2.4 MapReduce

MapReduce [25] is a high level large-scale data processing model that is used extensively in industry and academia. MapReduce has two main functions implemented by the user application: MAP and REDUCE. The MAP function is used for creating a list of key-value pairs and then allowing REDUCE function to do specific reduce operations on key-value pairs generated by the MAP function. It was introduced by Google and has several industry implementations such as Apache Hadoop. MapReduce is a distributed scheme, and normally fault-tolerance is achieved through replication of data into neighboring nodes and rescheduling individual tasks after node failures.

2.2.5 Parallel Virtual Machine

Parallel Virtual Machine (PVM) [26] was designed to enable a collection of heterogeneous computers interconnected by networks to be used as a coherent and flexible simultaneous computational resource. It provides a straightforward and general interface for the programmer. PVM supported both shared-memory and message passing paradigm of communication. It also included the support for failure detection and notification. Although PVM was a step towards the modern trends in distributed processing, it has been replaced by the introduction of MPI Standard.

2.3 Characterization of Applications

Characterization helps to understand application behavior and provides insight to focus on important design aspects. Both static and dynamic characterization is used to understand the use of MPI in large-scale applications.

While a number of surveys have reported on the use of general-purpose programming language in scientific and non-scientific programs [27, 28, 29] few studies have focused on the use of the MPI programming model. As far as we know, the first survey of MPI usage in scientific codes goes back to 1997 [30]. This work evaluated the use of the MPI programming model and of other models, such as PVM in scientific codes. The goal of this study was not to understand the use of MPI functionality but to evaluate the impact of software complexity of these models and its implications to runtime performance.

More recently, a number of attempts has been made to understand the use of MPI better. A survey of MPI usage in the US exascale computing project (ECP) is presented in [31]. This study focused on applications of the US ECP projects only. The goal was to understand the applications needs around MPI usage and surveyed issues related to possible problems of ECP codes with respect to the MPI Standard, whether current codes expect to use MPI at exascale, and questions about the dynamic behavior of codes. Previous work have reported MPI calls usage in production programs in specific HPC systems and centers. Rabenseifner [32] reports MPI calls usage in a Cray T3E and SGI Origin2000 systems.

MPI provides a profiling interface (PMPI) that can be used to collect different information about a set of MPI routines. A number of profiling tools have been developed using PMPI interface to characterize MPI applications.

MPIP [33] tool gathers performance data of scalable applications using profiling, MPI tracing, and accessing hardware counters. This empirical data is used to recommend architectural enhancements of those applications.

TAU [34] framework also provides an MPI wrapper to intercept MPI calls and track various information for performance analysis.

Darshan [35] is a I/O characterization tool designed to continuously monitor application I/O behavior in petascale systems. This characterization could help predict the I/O needs of future extreme scale applications.

Autoperf [36] is a lightweight MPI monitoring tool. It is intended to monitor the MPI usage of all jobs on a system rather than profiling a specific execution. The MPI usage logs collected from Autoperf are used to provide key insights into the use of MPI in production.

In a more recent study, Laguna et al. [12] present the first large-scale study of the MPI usage of applications. This survey statically analyzed more than one hundred distinct MPI programs covering a significantly large space of the population of MPI applications. It focused on understanding the characteristics of MPI usage with respect to the most used features, code complexity, and programming models and languages.

2.4 Resilience in Large-Scale Systems

As HPC systems grow exponentially in scale, there is a growing concern of fault tolerance in such systems for long-running applications. As hardware and software failure may occur more frequently in scale while long-running parallel applications are being executed, the overall system reliability, serviceability, and availability (RSA) has become a major concern in such systems.

Fault Tolerance is the capability of a system (e.g., processor, network device) to continue correct execution after it encounters failures. Though the system continues to function, the overall performance may get affected. An efficient fault-tolerant system should not only minimize the failure-free overhead, but recovery operations should also be fast. There are several considerations (e.g., how to detect failure, how to notify others about failure) while designing a fault-tolerant system. In this section, we discuss different phases of fault tolerance.

2.4.1 Failure Detection

The first step of fault tolerance is to detect a failure. The purpose of a fault detector in distributed and parallel systems is to detect both local and remote failures. A *perfect fault detector* provides two guarantees— (a) *Accuracy:* no process is reported as failed until it has actually failed and (b) *Completeness:* all surviving processes are eventually notified about the failure [37].

Heartbeat [38] is a failure detector for quiescent reliable communication in asynchronous message passing systems implemented without timeouts in systems with process crashes and lossy links. Here, each node sends a signal to its target nodes at a certain interval. A process considers a remote process as failed if it does not receive a signal from that remote process for a certain amount of time.

Gossiping [39] is another failure detector for distributed systems. In gossip protocol, at every T_{gossip} time, each process increments its *heartbeat counter* (use for failure detection) and sends it to a randomly selected process. A process is considered failed, if the heartbeat counter has not increased for more than T_{fail} seconds.

2.4.2 Failure propagation and Consensus

Error propagation disseminates the information about failures to other peers. In MPI, a process may wait for a message from another failed process forever which may eventually cause deadlock. So, it's necessary to notify other peers about the detected failures. Also, failure propagation is one of the properties of a perfect fault detector (see Section 2.4.1). Another fundamental building block of fault tolerance in distributed computing is consensus. Consensus algorithm is used among peers to achieve a unique decision.

Two-Phase Commit [40] protocol uses a linear-scale consensus algorithm. One of the processes acts as a *coordinator* and others are called *participants*. In the first phase, the coordinator initiates a request for vote to all processes (except the failed ones) and they respond with either commit or abort. In the second phase, the coordinator decides (commit or abort) based on the responses and broadcasts it to all participants. However, if the coordinator fails before broadcasting the decision, then all participants would wait until the coordinator recovers. Hursey et al. [41] presented a modified log-scale Two-Phase Commit algorithm that removes the blocking requirement when the coordinator fails. Here, a *termination detection* algorithm is added where a participant can linearly ask other participants whether they have received the decision or not.

Three-Phase Commit [42] eliminates the need for blocking in Two-Phase Commit algorithm by adding another round of messages to it. Here, the coordinator broadcasts a ready for commit message before broadcasting the commit message which allows participants to make a uniform decision without blocking when the coordinator fails. If the coordinator fails before sending the ready message, the participants will abort after a timeout. One the other hand, if the coordinator fails after the ready message and before the actual commit, participants will commit on timeout. However, this algorithm adds more overhead as it includes another round of operations.

Paxos [43] is another consensus algorithm that scales as well as the Two-Phase Commit algorithm while being non-blocking. While other algorithms can only tolerate process failure, Paxos can also tolerate network partitioning. However, it has proven challenging to understand and develop it in practice [44].

2.4.3 Failure Recovery Models

The goal of fault tolerance is to return an application to a consistent, error-free state after a failure. Several models have been designed to make MPI applications fault tolerant. These models differ in their design, types of failures they address, implementation approaches, and underlying protocols. Next, we review related work on providing fault-tolerance abstractions to MPI programs.

2.4.3.1 Rollback Recovery

Rollback recovery attempts to restart the application from a previously saved state after a failure. It is the most commonly used recovery mechanism in HPC systems. The two major rollback recovery techniques are checkpoint/restart and message logging.

Checkpoint/restart. The most common instance of rollback recovery is synchronous checkpoint/restart (CPR). A checkpoint is a snapshot of the state of the process at a particular point so that it could be restarted in case of a failure. In traditional checkpoint/restart (CPR), the application periodically saves its state, and when a process fails, the job is killed and resubmitted by the user; the application then loads the last checkpoint and continues execution. For large-scale HPC systems, CPR is the fault-tolerance method of choice. However, all processes require to be in a global consistent state for checkpointing. This process of establishing a consistent state may cause *domino effect*—may force some of the processes that did not fail to roll back to an earlier checkpoint [45]. This domino effect may lead the application to its initial state. Different CPR methods have been studied for HPC systems.

Uncoordinated checkpointing allows each process to checkpoint independently which reduces runtime overhead during normal execution [46]. However, it increases the storage overhead as each process maintains multiple checkpoints. Also, it might be difficult to find a global consistent state and might lead to domino effect.

Coordinated checkpointing enforces each of the processes to synchronize their checkpoints. It's not prone to the domino effect. Storage overhead is also reduced as each process maintains only one checkpoint. However, it adds overhead as global checkpoint requires internal synchronization prior to checkpointing. CoCheck [47] provides transparent checkpointing to MPI applications using a coordinated checkpoint/restart mechanism. To avoid global inconsistencies and domino effect, all processes synchronously flush their in-transit messages before checkpointing. In case of failure, the entire MPI application restart from the last checkpoint. CoCheck sits on top of the MPI library and uses an existing single process checkpoint method.

Unfortunately, current automatic CPR techniques might not be realistic in future exascale systems. As it is anticipated that the Mean Time Between Failure (MTBF) will be smaller than the time required for checkpointing and restarting [3].

Multi-level checkpointing uses system storage hierarchy to achieve low cost checkpointing in large-scale HPC systems. At scale, the cost in time and bandwidth of checkpointing to a parallel file system becomes expensive. Multi-level checkpointing uses lightweight and faster checkpoints to handle the most common failures and uses parallel file system checkpoints only for severe and less common failures. Scalable Checkpoint/Restart (SCR) [48] and Fault Tolerance Interface (FTI) [49] are multi-level checkpoint libraries that use memory, Solid State Drive (SSD) storage, local disk, and parallel file system to save checkpoints. While SCR keeps the file interface abstraction, FTI provides a data structure abstraction, masking from the programmer how the data to be saved is actually managed by the library. Applications can achieve significant performance gains by reducing checkpoint overhead with the use of these multi-level checkpointing.

There are three main approaches to implement checkpoint/restart.

In *Application-level* implementations, the checkpointing activities are carried out by the application. Basically, the application becomes responsible for saving checkpoint in persistent storage and restarting the application from a checkpoint after failure. One of the major benefits of this approach is that it can reduce checkpoint overhead by storing only the information necessary for restating the application (small checkpoint size). However, it lacks transparency and requires the programmer to have a good understanding of the application. Cornell Checkpoint (pre)Compiler (C3) [50] is an application-level checkpoint implementation where the application defines a set of directives and C3 replaces these directives with necessary checkpoint/restart functions in compile time.

In *User-level* implementations, checkpoint/restart is implemented in user-level libraries and linked to the application. This approach is not transparent to user as the application needs to be modified for linking to the checkpointing libraries. Libckpt [51] is an example of user-level checkpoint implementation. Distributed MultiThreaded CheckPointing (DMTCP) [52] is another example of user-level checkpoint that supports both traditional high-performance applications and typical desktop applications. DMTCP supports a transparent way to checkpoint with no re-compilation and re-linking of user binaries. It is agnostic to the underlying message passing library and does not require kernel modification. However, DMTCP is not network agnostic. A recently introduced checkpoint/restart method, MANA (MPI-Agnostic Network-Agnostic Transparent Checkpointing) [53], supports all combinations of the many MPI implementations and underlying network libraries. It uses a *split-process* approach, which separates the process memory into two halves. The lower half memory is associated with the MPI library and dependencies, while the upper half is associated with the MPI application's code. Only the upper half memory is saved or restored during checkpoint and restart. At restart time, the lower half can be replaced with new MPI libraries. To maintain consistent handles across checkpoint/restart, MANA records any MPI calls that can modify the MPI states and recreates those states by replaying the recorded calls during restart.

System-level checkpointing is implemented in either hardware or kernel. It is transparent to the application and no modification of application is required. Usually, a snapshot of the full-system is saved in a checkpoint. So, in large scale systems it can be impractical because of the checkpointing size. Also, system-level checkpointing is not portable to other platforms. Berkeley Lab Checkpoint/Restart (BLCR) [54] is a system-level CPR library. Several MPI implementations, including Open MPI, MPICH, LAM/MPI, added BLCR support.

Message Logging. In log-based rollback recovery, sent and received messages by each process are recorded in a log along with the checkpoint. The recorded message log is called a *determinant*. In case of a process failure, it can be recovered using the checkpoint and reapplying the logged messages. Typically, there are three types of message logging protocols.

Pessimistic logging synchronously logs the determinant to stable storage before it is allowed to affect the computation [55]. However, synchronous logging incurs a performance overhead as process is blocked while it logs determinant. This overhead can be lowered by using fast non-volatile memory for stable storage [56]. Bouteiller et al. [57] provides a model of pessimistic message logging protocol for MPI applications which reduces the overhead of message logging when no failure occurs by reducing the intermediate message copying.

Optimistic logging asynchronously logs the determinant to stable storage [58]. It assumes that the logging will complete before a failure occurs. Optimistic logging incurs little overhead in fault-free mode as it does not block the application while logging determinants from volatile storage to stable storage. However, it complicates the recovery process as volatile storage may lose the content.

Casual message logging has the advantages of both pessimistic and optimistic message logging [59]. Here, processes piggyback their non-stable determinants to other processes while sending messages. Only the most recent checkpoint on stable storage is required for rollback which reduces the storage overhead. However, it requires complex recovery protocol.

A quantitative assessment on different fault tolerance protocols shows that message logging will exhibit poor efficiency in extreme scale [60].

2.4.3.2 Replication-based Recovery

Replication is another strategy to achieve fault tolerance in distributed systems. However, it has been considered too costly for large HPC applications because of the increased amount of resource requirement.

N-version programming [61] is a replication-based method of building fault-tolerant software. It requires multiple independent implementations ("N" versions) of a specification. These versions run in parallel on a similar environment with identical inputs and produce its version of the output (ideally the same). The system uses the outputs of the majority, in case the results differ.

Practical Byzantine-fault-tolerant [62] method describes an algorithm to survive Byzantine faults in asynchronous systems using active replication (state machine replication). This method replicates the state machine across different nodes in a distributed system. It requires a minimum of $3f + 1$ replicas where f is the maximum number of replicas that may be faulty. Byzantine-fault-tolerant algorithms can allow a system to continue to work correctly even when there are software errors.

P_AR_{EP}-MPI [63] is an MPI implementation that uses process replication to support fault tolerance in exascale systems by utilizing the MPI profiling layer (PMPI). It supports proactive fault tolerance using partial replication of a set of application processes. $\mathrm{P}_A\mathrm{R}_{EP}$-MPI adaptively changes the set of replicated processes based on failure predictions. It also minimizes the resource wastage by keeping the number of replica processes to a minimum.

Intra-parallelization [64] is another process replication based fault tolerance model for MPI HPC applications. This model does not rely on failure prediction. Intra-parallelization replicates all processes instead of partial replication. When an MPI processes is replicated, both communication and computation steps are replicated on all replicas. Intra-parallelization introduces work-sharing between the replicas by avoiding full replication of computation steps. Instead, a computation step is divided into tasks; only one replica executes a task and sends the result to other replicas.

2.4.3.3 Application-level Recovery

Application-level recovery involves application user in the recovery process. The recovery model returns control to the application and lets the application react to a failure. Application-level failure recovery for MPI requires significant application changes and extension of the MPI interface.

Fault-Tolerant MPI (FT-MPI) [65] is one of the notable efforts to achieve fault tolerance for MPI applications by providing application programmers with different methods of dealing with failure rather than just using checkpoint/restart. Typically, application detects an error by checking the return code of any MPI call. After a failure, FT-MPI lets the user rebuild any communicators by—(a) SHRINK: removes failed processes and creates a new shrunk communicator which may change the rank of the processes, (b) BLANK: keeps gap in the communicator which creates invalid processes in the communicator, (c) REBUILD: creates new processes to replace the failed ones, or (d) ABORT: forces a graceful abort of the application. It also allows various modes of communication after failure—(a) NOP: returns an error code for any communications and (b) CONT: continues communication for nodes not affected by failures. FT-MPI is not transparent to the application and increases code complexity.

Run-Through Stabilization (RTS) [66] is another proposal to introduce failure handling to MPI. Run-Through Stabilization allows the application to continue its execution even with fail-stop process failure. It introduces new construct to MPI and provides a *perfect* failure detection for *fail-stop* process failure via error handlers. In order to continue execution, it introduces the ability to *validate* communicators when failure occurs and allows the application to use these communicators. Unfortunately, this introduces implementation complexity and was not adopted in the MPI standard.

User Level Failure Mitigation (ULFM) [67, 68, 69] is an approach of fault tolerance for MPI applications proposed by the Fault Tolerant Working Group of the MPI Forum. ULFM supports fault tolerance in MPI through a set of interfaces (API extension) exposed to the user. To recover from a process failure, it creates a new communicator (not replace) by excluding the failed processes and continue execution. Figure 2.2 shows an overview of the mechanism to

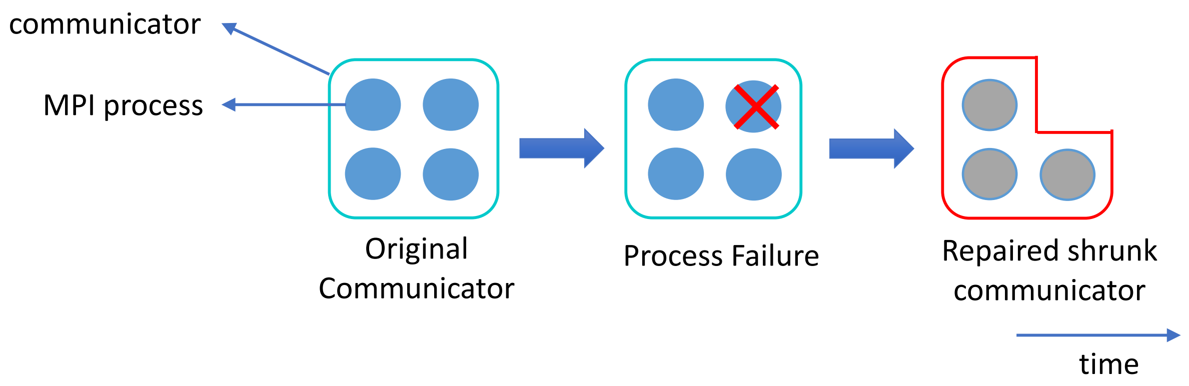


Figure 2.2: User Level Failure Mitigation (ULFM): After a process failure, the communicator is repaired by shrinking it to remove the failed process and the application continues its execution thereafter.

repair the faulty communicator in ULFM. The three basic operations of ULFM are— notifying application about failure, propagating error to all processes, and recovering from the failure. It masks all failures as *process failure*. This approach uses in-band error codes to notify the application of failures. The application checks the error code returned by MPI routines to receive notification of failures. MPI routines return the error code `MPI_ERR_PROC_FAILED` to indicate process failure (elsewhere), and the knowledge of process failures is local to any process that receives the error code. To prevent infinite deadlocks on communication operations with failed remote processes, ULFM introduces `MPI_COMM_REVOKE`. It is a non-local and non-collective operation that is used to propagate failure information throughout a MPI communicator. Once the failure notification is disseminated, all live processes use `MPI_COMM_FAILURE_ACK` and `MPI_COMM_FAILURE_GET_ACKED` to get the group of actual failed processes. This group of failed processes is excluded from the continuing application. To recover from a failure, once a MPI communicator has been revoked, the remaining live processes call the collective operation `MPI_COMM_SHRINK` to create a new shrunken communicator. In a fault-tolerant application, sometimes processes need to agree on certain value using an agreement algorithm. ULFM provides `MPI_COMM_AGREE` to perform a fault tolerant agreement algorithm over a boolean value among all alive processes. ULFM supports MPI file and one-sided objects using similar functionality. Several fault-tolerant models have been proposed using ULFM. Local Failure Local Recovery (LFLR) [70] is one such model that uses spare processes after failure to keep the number of processes constant. Thus, LFLR eliminates the complexities of load balancing across fewer application processes.

2.4.3.4 Transactional Resilience Scheme

Fault-Aware (FA-MPI) is a lightweight, transaction-based fault tolerance model for MPI [71, 72]. With this transaction-based model, an application can choose to use FA-MPI to achieve a fine-grain fault tolerance model by encapsulating every MPI operation in a single transaction. Or, should the application want to balance performance with fault tolerance, the application can choose to instead put many MPI operations into a single transaction. As FA-MPI is designed to be an extension of the MPI API, the application can use the fault awareness provided by FA-MPI to determine the level of fault tolerance it wants. It allows the applications to implement a wide range of fault-tolerant methods. The application can use checkpoint/restart, forward, or backward recovery.

A fundamental building block of this model is a TryBlock, which allows a series of operations to be *tried*. It commits when all operations succeed and enables an application to roll *backward* or *forward* when some operations fail. TryBlock transactions can be local in scope—where only the local process must decide what to do—or global in scope—where all failures are synchronized among all processes. FA-MPI incorporates timeouts into its fault detection methods to prevent the possibility of deadlock introduced when dealing with fault tolerance. These timeouts can be user defined. A method for communicator recovery with the same number of original processes as prior to the failure is also provided in complement to the TryBlock transactional framework. Finally, FA-MPI is not restricted only to addressing process failure.

2.4.3.5 Global-Restart

This approach provides rollback recovery type of resilience for MPI applications. The idea behind global-restart is to provide a simple mechanism to restore application state after a failure to a previously stored state saved as a checkpoint and then resume computation from that state without killing the entire MPI job, a key goal for production MPI programs seeking resilience. In global-restart, applications do not need to perform steps to detect and propagate failures,

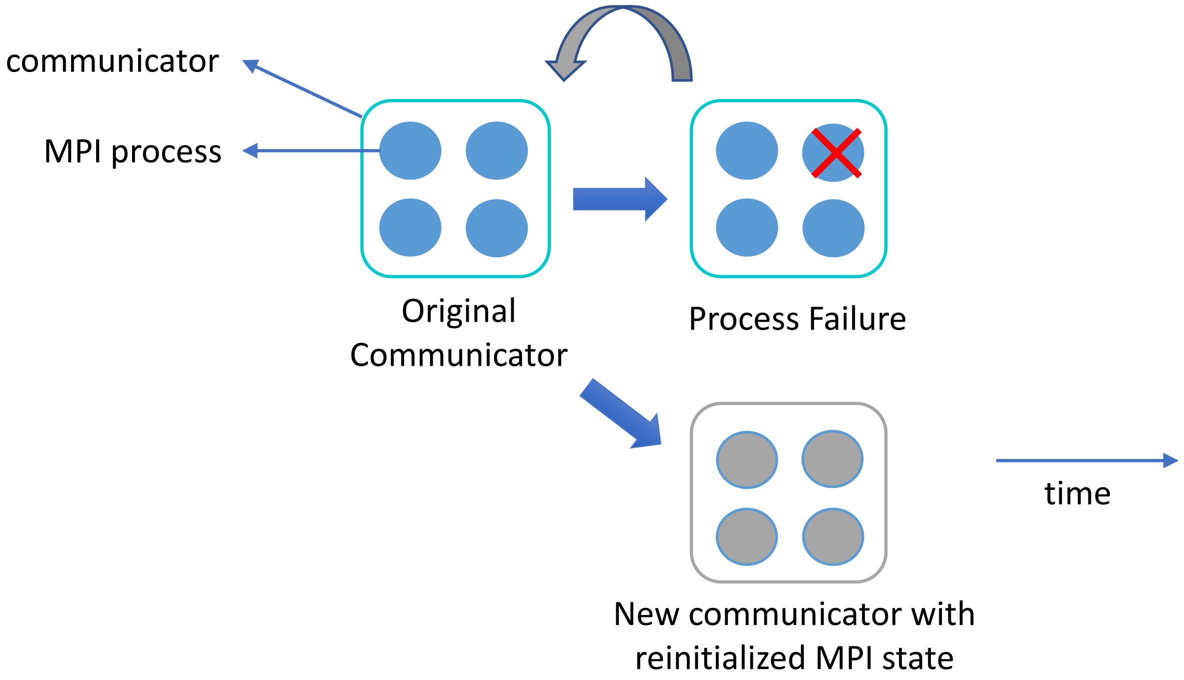


Figure 2.3: Reinit: After a process failure, the application is rolled back to the last checkpoint and continues execution from there after the MPI state is reinitialized.

and to recover the MPI state (e.g., communicators); these steps are performed by external components.

Reinit [73] is one of the implementations of this model. Reinit is a non-shrinking recovery where application resumes execution with the same number of processes as it had prior to the fault. Here, upon detecting a fail-stop process failure, new processes are used to replace those that have failed. Figure 2.3 shows the overview of Reinit recovery mechanism. This model assumes a perfect fault detector within MPI runtime. Thus, the application need not check for faults. However, it provides a routine `MPI_FAULT`, which the application can use to notify MPI about a failure. Also, the propagation of failure is done automatically within the MPI implementation. The failure propagation is triggered as soon as MPI runtime detects a failure without waiting for message completion at the receivers. Then, it calls the cleanup handler to clean any application or library specific states and finally, re-initializes the MPI processes. It introduces a new routine `MPI_REINIT` for the reinitialization. This routine calls a restart handler (a pointer to a user-defined function) that serves as the starting point of a resilient application. After the reinitialization, the resulting state of the processes is the same as the

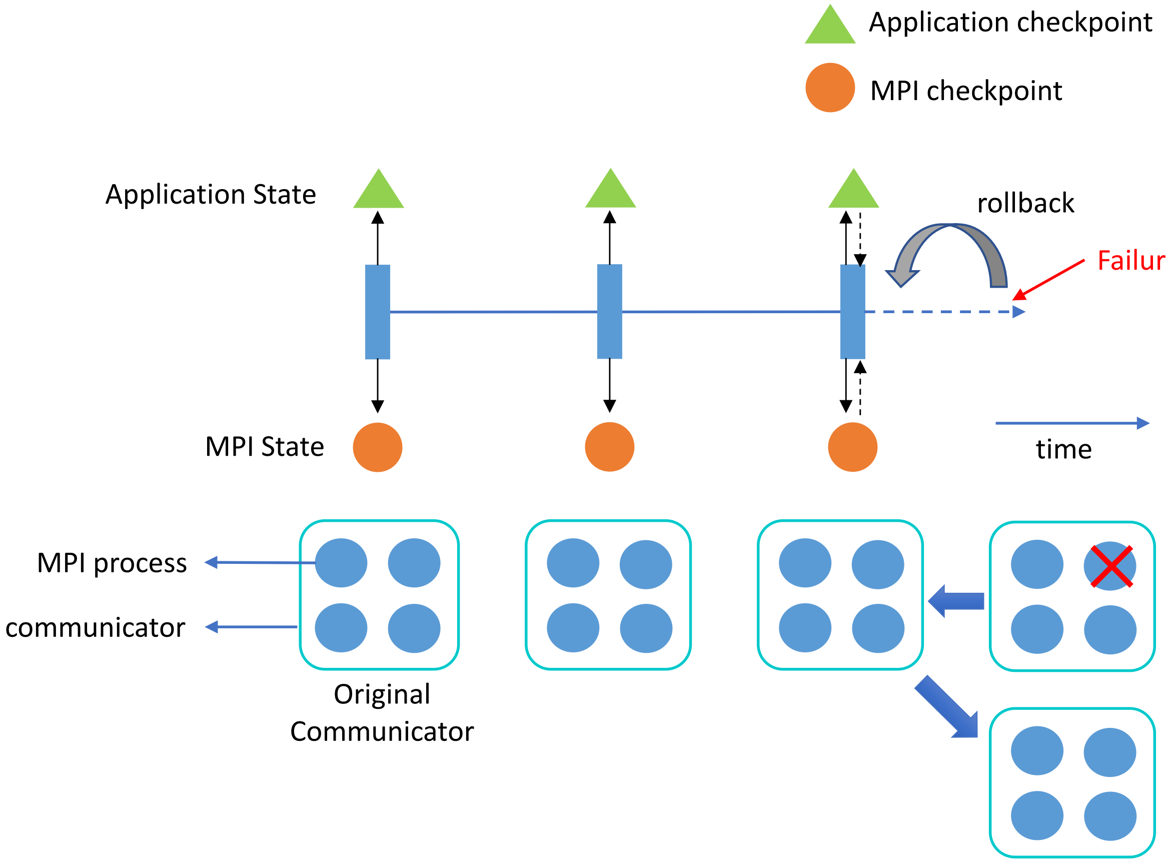


Figure 2.4: Overview of MPI Stages. The MPI state is checkpointed at the same instance of the application checkpoint. At a failure, both MPI and application states are restored by reloading the most recent checkpoints.

state after `MPI_Init()` returns. The recovery state of an MPI process is defined by— (a) NEW: first time start process, (b) RESTARTED: the process that has been restarted due to a fault, and (c) ADDED: the process that has been added to the existing job. To jump to the restart handler from any point of the execution, Reinit use setjmp/longjmp semantics of C language. The setjmp/longjmp semantics is language specific and not supported in Fortran.

Fenix [74] is another example of global-restart model that makes use of the ULFM interface. It implements a non-shrinking recovery and transparently spawned a new process to repair the communicators. The error codes returned after a failure are detected using MPI's profiling interface. Once the communicators are recovered, a long jump is used to return execution to `FENIX_INIT` which is configured to resume to the beginning of execution.

2.4.3.6 Our Approach—MPI Stages

The closest competitor to our approach, MPI Stages, is the global-restart approach, Reinit. While the global-restart approach reduces the recovery time of bulk synchronous applications by eliminating the need of terminating the job upon a failure, processes are restarted from the beginning of the program even though they did not experience a failure. This is required to allow the replaced process to build its own MPI state (e.g., communicators and groups), along with the live processes before they reach the point in the computation where the failure occurs. As shown in Figure 2.4, the core idea of our approach is to be able to checkpoint the MPI state at similar intervals as the application checkpoints. Upon a failure, the MPI state is rolled back to the latest checkpoint and is updated with the most recent and consistent values. The key difference of our approach and the Reinit method is that our approach eliminates the need of the processes to participate in rebuilding MPI state—MPI Stages saves a checkpoint of the MPI state, which is then used to fully restore the state of the failed process. When a failure is detected by MPI runtime, the failure is notified to all live processes which then discard their current work and rollback to a previous checkpoint. This is similar to a collective operation in the communicator except the failed process. Our recovery model discards the MPI states from the last synchronous checkpoint to the failure. Any messages in transit across processes are lost and will need to be re-sent. Chapter 5 describes in detail the MPI Stages approach.

Chapter 3

Understanding the use of MPI in exascale HPC applications

3.1 Introduction

The Message-Passing Interface (MPI) is an extensively used programming model for multi-node communication in scientific computing. Many large-scale HPC applications are successfully using MPI in petascale systems and will continue the use in exascale systems [75, 76, 6]. To increase the applicability of MPI in next-generation exascale systems, the MPI Forum is actively working on updating and incorporating new constructs to MPI Standard. Thus, understanding the state-of-the-practice of MPI usage is extremely important to the HPC community. A detailed understanding of MPI usage is required to optimize the communication characteristics of applications. The MPI usage statistics can help the MPI Standardization body to prioritize the standardization of new features. Also, the MPI implementers and HPC centers can allocate resources to optimizing the most commonly used MPI features from a detailed MPI usage characterization.

Here, we focus on understanding the characteristics and communication patterns of MPI in exascale applications. The Exascale Computing Project (ECP) is an effort to accelerate the development and delivery of a practical exascale computing ecosystem [77]. To achieve an exascale-capable ecosystem, ECP focuses on three areas—application development, software technology, and hardware technology. Both application development and software technology include various projects related to MPI. Some of the MPI-related ECP projects specifically focus on developing high-performance MPI implementations and enabling applications to effectively use MPI to scale to future exascale systems.

Most ECP applications are typically large and complex with thousands to millions of lines of code. As a means to access their performance and capabilities, most of these applications have representative "mini-apps" to serve as their proxies or exemplars. A proxy application is designed to represent key characteristics of the real application without sharing the actual details. As part of ECP, multiple proxy apps have been developed. The ECP proxy apps suite consists of proxies representing the key characteristics of the exascale ECP applications [78].

We analyze 14 ECP proxy apps using both profiling and static analysis. We analyze three important characteristics of the apps: (a) most used MPI features and MPI routines, (b) communication characteristics, and (c) use of multi-threaded environment.

The key findings from our analysis are as follows:

(1) A large proportion of the applications surprisingly do not use advanced features of MPI. We find, for example, that point-to-point (blocking and non-blocking) routines are more prominently used than persistent point-to-point or one-sided routines.

(2) The majority of the applications use only a small set of features from the MPI Standard—a considerable number of applications use only the point-to-point and collective communication features of the standard, leaving other parts of the standard totally unused.

(3) MPI collectives occupy a significantly larger portion of the communication time compared to point-to-point. `MPI_Allreduce` is the most frequently used collective call. We also find that applications mostly use small messages for collective operations.

(4) We observe that about 1/2 of the applications use a mixture of MPI and OpenMP programming models. However, none of the applications use `MPI_THREAD_MULTIPLE` environment. The reason might be the performance implication of MULTIPLE in MPI implementations.

3.2 MPI Features Classification

In this section, we provide a summary of the key MPI features we analyze in the proxy applications.

The MPI Standard version 3.1 (the most recent version) specifies roughly 443 distinct routines. We grouped these routines into 11 categories. In the analysis, we refer to these categories as MPI *features*. We summarize these categories as follows—for more information, the reader should refer to the MPI Standard [15]:

- **Point-to-Point communication**: This feature specifies how to transmit messages between a pair of processes where both sender and receiver cooperate with each other, which is referred to as "two-sided" communication. To communicate a message, the source process calls a send operation and the target process must call a receive operation. MPI provides both blocking and non-blocking forms of point-to-point communications.

- **Collective communication**: This feature describes synchronization, data movement, or collective computation that involve all processes within the scope of a communicator. All processes of the given communicator need to make the collective call. Collective communications do not use tag. Collective communications will not interfere with the point-to-point communications, and vice versa. It has both blocking and non-blocking variants.

- **Persistent communication**[1]: To reduce the overhead of repeated point-to-point calls with the same argument list, this feature creates a persistent request by binding the communication arguments and uses that request to initiate and complete messages.

- **One-Sided communication**: This feature defines a communication where a process can write data to or read data from another process without involving that other process directly. It is known as Remote Memory Access (RMA) communication. The source process can access the memory of the target process without the target's involvement. The source is limited to accessing only a specifically declared memory area on the target called a "window."

- **Derived Datatype**: This feature describes how to create user-defined structures to transfer heterogeneous/non-contiguous data. It allows different datatypes to be transferred in one MPI communication call.

[1] Persistent Collective communication, which is part of MPI-4, is not considered in this study.

- **Communicator and Group management**: A *Group* is an ordered set of processes and a *Communicator* encompasses a group of processes that may communicate with each other. This feature describes the manipulation of groups and communicators (e.g., construction of new communicator) in MPI.
- **Process Topology**: An MPI application with specific communication pattern can use this feature to specify a mapping/ordering of MPI processes to a geometric shape. Two topologies supported by MPI are Cartesian and Graph.
- **MPI I/O**: This feature provides support of concurrent read/write to a common file (parallel I/O) from multiple processes.
- **Error Handling**: An MPI application can associate an error handler with communicators, windows, or files. If an MPI call is not successful, an application might use this error handling feature to test the return code of that call and execute a suitable recovery model or abort the application.
- **Process management**: This feature specifies how a running MPI program can create new MPI processes and communicate with them.
- **Tools Interface**: This interface allows tools to profile MPI applications and to access internal MPI library information.

There are also miscellaneous features, such as `MPI_Wtime/MPI_Wtick`, and generalized requests not specific to any of the above categories. We omit these miscellaneous features from our analysis.

3.3 Data Collection

We use both static and runtime analysis to collect MPI usage data. We used a grep-based script to gather MPI routines and features usage data. To analyze the communication characteristics of these applications, we implemented a profiling tool to collect data using the PMPI interface. Instead of using an existing profiling tool (e.g., MPIP [33], TAU [34]), we implemented our own profiler. We use this new profiling tool to collect only the required information needed for our analysis. We log the use of tag and communicator in addition to the message size and time spent in each call in our profiler. The profiler is written in C++. The LD_PRELOAD environment variable is used to inject the profiler at runtime via the dynamic linker.

The profiler collects various MPI usage data from a running MPI job and stores them in log files. These log files indicate all the MPI routines called by the application, the time spent in each routine, and the number of bytes sent or received by all communication routines.

The log files are generated at `MPI_Finalize`. The profiler collects data for all MPI processes. However, to reduce the number of files, we only generate two log files. The first one is for the process with MPI process rank 0 and the other one is for the process with MPI rank $\frac{N}{2}$ (middle rank), where N is the size of `MPI_COMM_WORLD`.

The profiling tool implements an MPI wrapper for each MPI routine. Currently, the MPI profiler only supports those MPI routines required by the ECP proxy applications. In each such MPI wrapper, the wrapper first determines whether the call is user MPI routine or a call generated from inside of a collective operation. For each user routine, it registers the datatype and the count of the exchanged type that is used to calculate the bytes sent or received. Then it records the start time using `MPI_Wtime`. The wrapper then calls the corresponding PMPI routine. After returning from the PMPI call, the wrapper collects the end time. Finally, for each user routine, it stores the routine name, datatype, count, time spent in the call, source, tag, and communicator when applicable. Algorithm 1 presents pseudocode of the `MPI_Send` wrapper.

Algorithm 1 Pseudocode for MPI Wrapper

```
procedure MPI_Send(. . .)                                        ▷ MPI wrapper routine
    Determine whether user call or not
    $t1 \leftarrow MPI\_Wtime$
    $err \leftarrow PMPI\_Send$                                      ▷ Call normal MPI
    $t2 \leftarrow MPI\_Wtime$
    if $user\_call$ then
        Record the call information    ▷ Number of byte sent, time spent in call (t2-t1)
```

Profiler Overhead. One of the most important factors of profiling is the overhead associated with it. The profiler should not add significant overhead to the execution time as compared to the non-profiled execution. To analyze the overhead added by our profiler, we calculate the

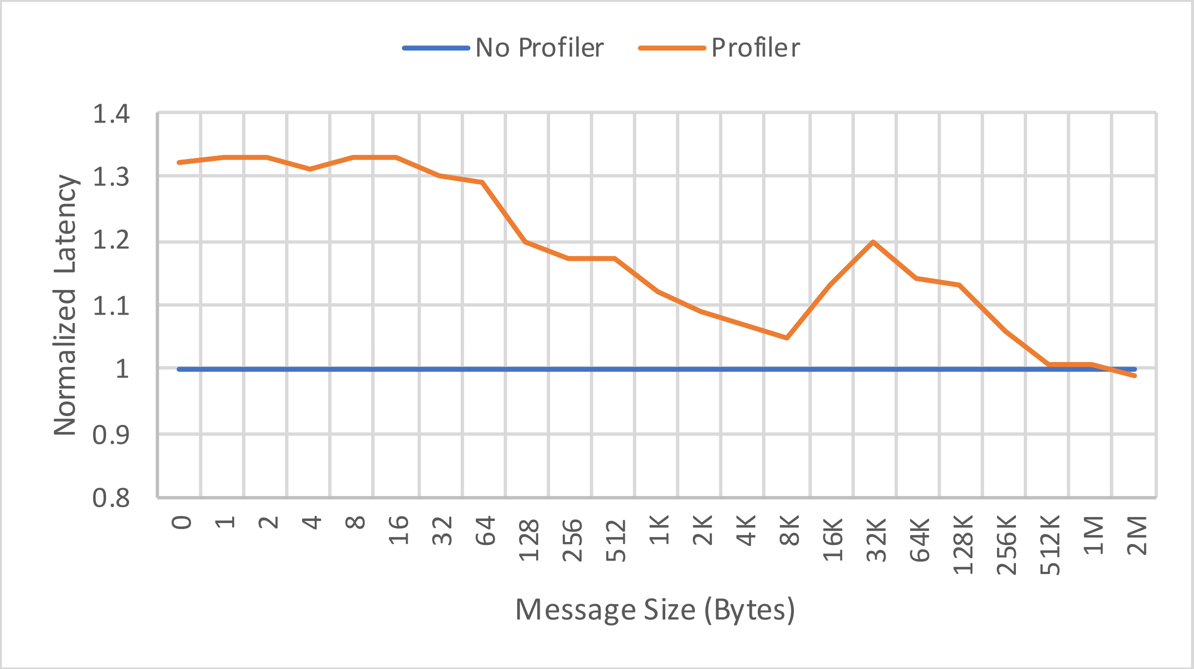


Figure 3.1: Overhead added by profiling using ping-pong benchmark latency. We observe that the overhead reduces as we increase the message size.

latency of a ping-pong benchmark with and without the profiler. Figure 3.1 shows the overhead associated with profiling. Our profiler adds almost 30% overhead for small message sizes. The overhead starts to decrease as we increase the message size. For large messages (¿=256K), the overhead appears to be negligible.

Limitations. Our profiling tool only intercepts the C language binding of MPI. It does not intercept the Fortran API. Only two of the applications (PICSARlite and thornado-mini) from ECP suite use the Fortran API. Thus, we did not collect profiling data for these two applications. One possible solution is to extend the current wrapper to intercept Fortran API of MPI as part of future work.

3.4 Overview of Applications

In this study, we analyzed the ECP Proxy Applications Suite 2.0[2] [78]. The suite comprises 15 proxy applications. These proxies represent different key characteristics (e.g., performance, communication pattern) of ECP applications. Since ECP applications target future exascale

[2] Released on October 1, 2018

Table 3.1: Application overview

Application (language)	**Description**	**Third-party Library**	**Model**
AMG(C)	Parallel algebric multigrid solver for linear systems arising from problems on unstructured grids	hypre	MPI + OpenMP
Ember (C)	Represent highly simplified communication patterns relevant to DOE application workloads	N/A	MPI
ExaMiniMD (C++)	Proxy application for Molecular Dynamics with a Modular design	N/A	MPI
Laghos (C++)	Solves Euler equation of compressible gas dynamics using unstructured high-order finite elements	hypre MFEM Metis	MPI
MACSio (C)	Multi-purpose, scalable I/O proxy application that mimics I/O workloads of real applications	HDF5 Silo	MPI
miniAMR (C)	Applies a 3D stencil calculation on a unit cube computational domain– divided into blocks	N/A	MPI + OpenMP
miniQMC (C++)	Designed to evaluate different programming models for performance portability	N/A	MPI + OpenMP
miniVite (C++)	Detects graph community by implementing Louvain method in distributed memory	N/A	MPI + OpenMP
NEKbone (Fortran)	Thermal Hydraulics mini-app that solves a standard Poisson equation	N/A	MPI
PICSARlite (Fortran)	Portrays the computational loads and dataflow of complex Particle-In-Cell codes	N/A	MPI
SW4lite (C, Fortran)	Solves the seismic wave equations in Cartesian coordinates	N/A	MPI + OpenMP
SWFFT (C)	Run 3D distributed memory discrete fast Fourier transform	FFTW3	MPI
thornado-mini (Fortran)	Solves radiative transfer equation in a multi-group two-moment approximation	HDF5	MPI
XSBench (C)	Represents a key computational kernel of the Monte Carlo neutronics application	N/A	MPI + OpenMP

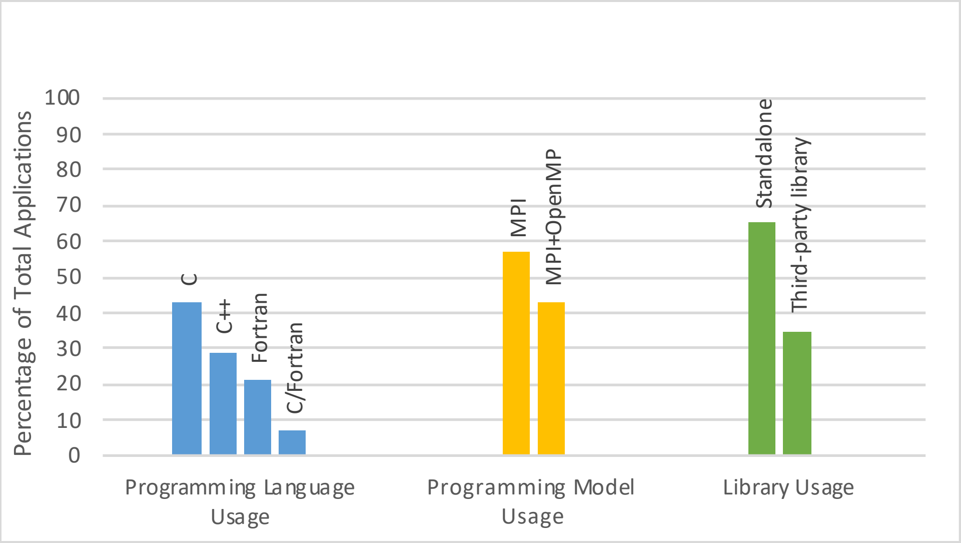


Figure 3.2: Illustration of programming language, model, and library usage. We observe that almost half of the applications use MPI+OpenMP model and C/C++ is most used language.

systems, we chose the ECP proxies as our test corpus to understand the usage of MPI in prospective exascale applications. We identified 14 applications in the suite that use MPI.

Table 3.1 provides an overview of the applications described in this study. The table presents the use of programming language, model, and third-party library along with a short description of each applications.

In addition to the direct usage of MPI, it is also interesting to note which languages and programming models are used in MPI applications. Figure 3.2 shows the usage of programming languages and models in ECP proxy applications. As shown, C and C++ are the predominant programming languages among these applications. C comprises 43% of the language usage. C++ comprises 29%, and Fortran at 21% of the application code bases. The usage of C++ is through the MPI C interface because no C++ MPI interface exists[3].

Figure 3.2 also shows the use of hybrid programming model in our sample applications. Here, the hybrid model comprises of MPI and OpenMP, where OpenMP corresponds to multi-threaded programming model used along with MPI for in-node parallelism. We find that 43% of the applications use the hybrid (MPI+OpenMP) programming model.

Table 3.1 shows that almost two-thirds of the proxies are standalone applications. About 35% of the applications use a number of third-party software libraries. However, not all of

[3] No C++ interface exists after MPI 2.1

these third-party libraries are MPI-based. Four of the applications—AMG, Laghos, MACSio, and thornado-mini use MPI-based third-party numerical and I/O libraries. One of the MPI-based libraries, hypre [79], uses the hybrid programming model. Another library, HDF5 [80], uses pthreads as its parallel execution model.

3.5 Overview of MPI Usage in Applications

In this section, we overview the overall use of MPI in the proxy applications. To understand the usage of MPI, we categorize the MPI calls into MPI features. We group the MPI calls into 11 categories—*P2P*, *Collective*, *Persistent*, *One-Sided*, *Datatype*, *Topology*, *Tool*, *Comm_Group*, *Error*, *File* (MPI I/O), and *Process_Management* based on the feature classification described in Section 3.2. We analyze the usage of MPI for each of the features. Along with these features, we also analyze the initialization of MPI. Finally, all the environmental management (e.g., memory allocation) and language bindings routines are considered as *Other* category and we do not consider those in our analysis.

3.5.1 MPI Initialization

The MPI execution environment is initialized by using `MPI_Init` or `MPI_Init_Thread`. If `MPI_Init_Thread` is used, the program can use one out of four thread environment options: `MPI_THREAD_SINGLE`, `MPI_THREAD_FUNNELED`, `MPI_THREAD_SERIALIZED`, and `MPI_THREAD_MULTIPLE`. We find that 43% of the applications in our sample use the hybrid (MPI+OpenMP) programming model (as shown in Figure 3.2). However, not all of these programs use `MPI_Init_thread` to initialize the thread environment.

The routine `MPI_Init_thread` takes two parameters: *required* which indicates the level of desired thread support and *provided* indicating the actual thread support while executing the program. In our static analysis, we only capture the *required* thread level specified by the application.

Our analysis shows that almost all of these applications (90%) use `MPI_Init` for initialization. Only one application (PICSARlite) uses MPI thread-based initialization. However, in PICSARlite, the required level of thread support is `SINGLE` and `FUNNELED`. Although, roughly half of the proxy applications use the hybrid programming model; however, none of them use the thread MULTIPLE mode. One possible reason of not observing MULTIPLE might be that the performance implications of MULTIPLE in MPI implementations.

3.5.2 MPI Communication

Here, we analyze the MPI features for communication—point-to-point (P2P), collective, persistent, and one-sided operations. For each application, we statically collect all the MPI communication calls. Then we analyze the communication characteristics (e.g., payload distribution, communication and computation time) of each application using the profiler mentioned in Section 3.3.

3.5.2.1 Communication Calls

Table 3.2 presents the significant MPI calls used for communications in the applications including the ones used in the third-party libraries. The table includes all the MPI communication calls (P2P and collective) found in the application source code. During a specific run, the application may use a subset of these calls for communications.

Two of the proxy applications (AMG and PICSARlite) use persistent point-to-point routines — MPI_Send_init and MPI_Recv_init. The application, Laghos, uses persistent routines through its libraries. Our results also show that only one application (miniVite) uses one-sided (RMA) routines — MPI_Put and MPI_Accumulate.

Two of the applications (miniQMC and XSBench) do not use any point-to-point MPI calls. These applications primarily investigate in-node parallelism issues using OpenMP. In XSBench, there is only one call of MPI collective routines (a barrier and a reduce) at the end to aggregate results.

Most of the applications use point-to-point and collective operations for communications rather than persistent and one-sided. Applications use both the blocking and non-blocking variants of point-to-point routines. For collective routines, applications mostly use the blocking variant.

Table 3.2: MPI calls used for communications

Application	**Point-to-point**		**Collective**	
	Blocking	**Non-blocking**	**Blocking**	**Non-blocking**
AMG, Laghos	MPI_Send MPI_Recv	MPI_Isend MPI_Irsend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Allgather{v} MPI_Gather{v} MPI_Alltoall MPI_Barrier, MPI_Bcast MPI_Scan MPI_Scatter{v}	N/A
Ember	MPI_Send MPI_Recv	MPI_Isend MPI_Irecv	MPI_Barrier	N/A
ExaMiniMD	MPI_Send	MPI_Irecv	MPI_Allreduce MPI_Scan	N/A
MACSio	MPI_Send MPI_Ssend MPI_Recv	MPI_Isend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Allgather MPI_Barrier, MPI_Bcast MPI_Gather, MPI_Scatterv	N/A
miniAMR	MPI_Send MPI_Recv	MPI_Isend MPI_Irecv	MPI_Allreduce MPI_Alltoall MPI_Bcast	N/A
miniQMC	N/A	N/A	MPI_Reduce	N/A
miniVite	MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Alltoallv MPI_Barrier, MPI_Bcast MPI_Exscan	MPI_Ialltoall
NEKbone	MPI_Send MPI_Recv	MPI_Isend MPI_Irecv	MPI_Allreduce MPI_Barrier, MPI_Bcast	N/A
PICSARlite	MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Allgather MPI_Barrier, MPI_Bcast	N/A
SW4lite	MPI_Send MPI_Recv MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Allreduce, MPI_Reduce MPI_Allgather, MPI_Gather MPI_Barrier, MPI_Bcast	N/A
SWFFT	MPI_Sendrecv	MPI_Isend MPI_Irecv	MPI_Allreduce MPI_Barrier	N/A
thornado-mini	MPI_Send	MPI_Isend MPI_Irecv	MPI_Allreduce MPI_Allgather MPI_Barrier, MPI_Bcast MPI_Scatterv	N/A
XSBench	N/A	N/A	MPI_Barrier MPI_Reduce	N/A

3.5.2.2 Communication Characterization

In this section, we analyze the communication characteristics of the proxy applications using the PMPI profiler.

System. We compile and run the proxy applications at the Catalyst cluster from Lawrence Livermore National Laboratory. The cluster has Intel (Xeon E5-2695) 24-core 2.4 GHz nodes, 128 GB of memory per node, and Infiniband QDR interconnect. The system is composed of 324 nodes with total 7776 cores. We use OpenMPI (3.0.1) as the MPI implementation with GCC (7.3.0) as the compiler for all applications.

Analysis. We ran each application with 256 processes using the minimal configuration required to execute it. We analyze three important communication characteristics of the apps: (a) message traffic in term of call count and time, (b) communication and computation time, and (c) payload distribution for both point-to-point (P2P) and collective operations. In our analysis, we do not include miniQMC and XSBench. In miniQMC, we found no MPI routines during the run although we found one `MPI_Reduce` in the application code during our static analysis. In XSBench, there is only one call to `MPI_Barrier` and `MPI_Reduce`, so we do not include it in our analysis.

Table 3.3 shows the actual MPI communication routines invoked by each of the application during our test run. It only shows the routines directly called by the application and not by any third-party library. The table shows that the applications use small subsets of MPI routines.

We characterize the message traffic in terms of number of P2P and collective calls. Figure 3.3 presents the percentage of total P2P and collective calls made by the process of rank 0 and the process of middle rank. In terms of call count, the communication is dominated by point-to-point calls. Only one application, SWFFT has nearly equal numbers of P2P and collective calls. In miniAMR, the process of middle rank performs three more collective operations than the process of rank 0. Here, the application creates a new communicator using `MPI_Comm_split` and performs collective operations in a subset of processes.

From the log files of our profiler, we collect the number of times each P2P and collective operations have been called and the total time spent on each type of operations. An MPI

Table 3.3: MPI calls invoked by application during a run

Application	Point-to-point								Collective							
	MPI_Send	MPI_Isend	MPI_Recv	MPI_Irecv	MPI_Sendrecv	MPI_Ssend	MPI_Send_init	MPI_Recv_init	MPI_Allreduce	MPI_Reduce	MPI_Barrier	MPI_Bcast	MPI_Scan	MPI_Alltoall	MPI_Alltoallv	MPI_Gather
AMG	×	×	×	×			×	×	×		×	×	×			
Ember	×	×	×	×							×					
ExaMiniMD	×			×					×				×			
Laghos	×	×	×	×					×	×	×	×	×			
MACSio			×			×			×	×	×	×				
miniAMR	×	×	×	×					×			×		×		
miniVite					×				×	×	×	×		×	×	
NEKbone	×	×		×					×		×	×				
SW4lite					×				×	×	×					×
SWFFT		×		×					×		×					

application performs many local operations (e.g., MPI_Comm_rank); we do not count these local operations in our analysis.

Figure 3.4 and 3.5 show the total runtime of the proxy applications by the process of rank 0 and the process with middle rank, respectively. The total runtime is the time between MPI_Init and MPI_Finalize. We breakdown the total runtime into computation and communication (P2P + collective) time. The communication time consists of point-to-point and collective communication time collected from the log file. The computation time is calculated by subtracting the communication time from total runtime. Figure 3.4a and 3.5a show the percentage of P2P, collective, and computation time for the rank 0 process and the middle rank process, respectively, whereas Figure 3.4b and 3.5b show the total time spent (in ms).

The majority of the applications spent more than 50% of their time in communication. In terms of time, communication is dominated by collective operations, although Figure 3.3 shows that P2P call count is much higher than collective call count. Our result shows that only the miniAMR application spent more time in P2P rather than collective. For ExaMiniMD, the middle rank process spent more than 50% of its communication time in P2P.

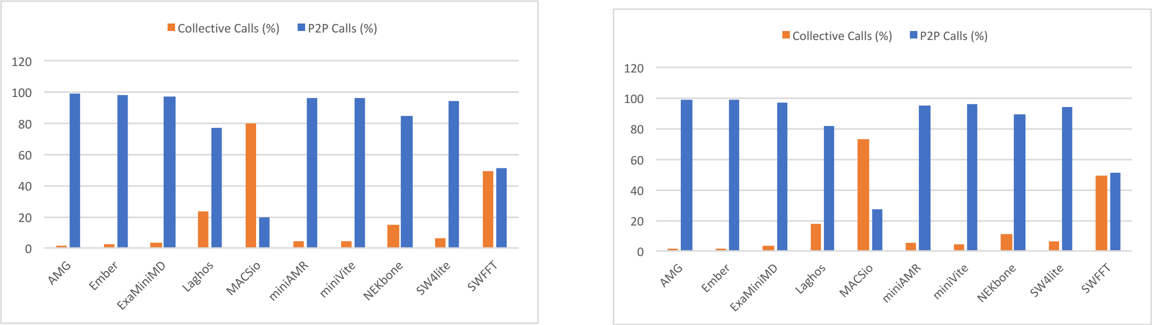
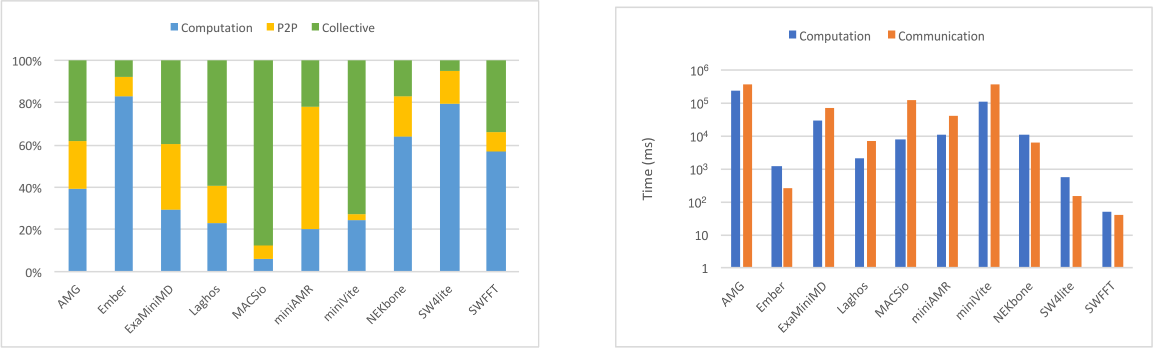


Figure 3.3: Illustration of total number of point-to-point (P2P) and collective calls made by the rank 0 process (left) and the middle rank process (right). We observe that communication is dominated by P2P in terms of call count.



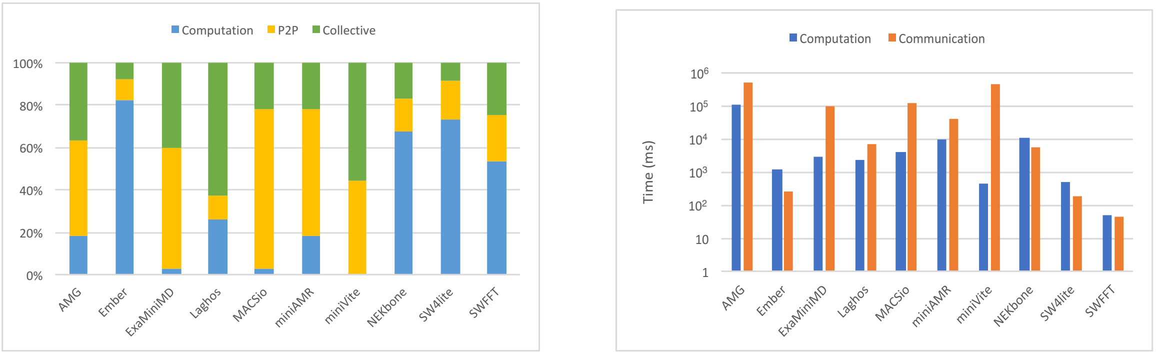
(a) Percentage of P2P, collective and computation time.

(b) Time spent in communication (P2P + collective) and computation.

Figure 3.4: Total time spent in communication (point-to-point and collective) and computation by the rank 0 process. We observe that majority of applications spent more time in communication and communication time is dominated by collective operations.

MPI Point-to-Point Operations. Figure 3.6a and 3.7a show the aggregated call count of the point-to-point call for the rank 0 process and the middle rank process, respectively. The non-blocking communication calls (`MPI_Isend` and `MPI_Irecv`) are used more frequently than the blocking calls (`MPI_Send` and `MPI_Recv`). Here, we do not include `MPI_Ssend` as it has been invoked only 20 times by MACSio.

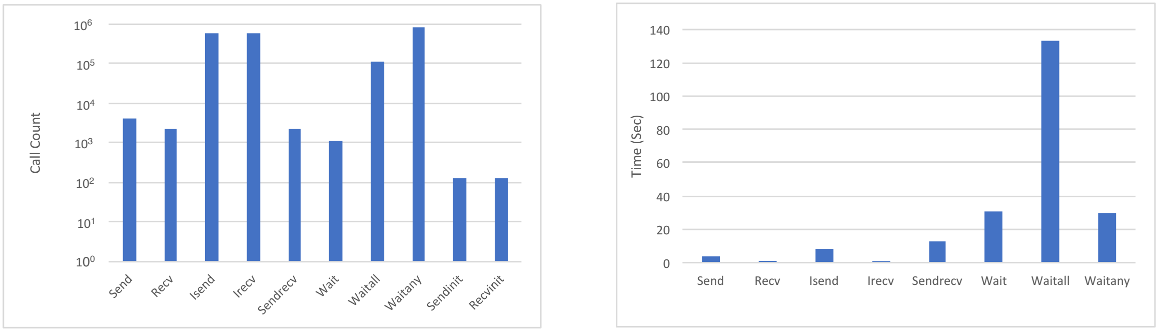
The aggregated time spent in each point-to-point operation in all proxy applications for the rank 0 process and the middle rank process is shown in Figure 3.6b and 3.7b, respectively. The `MPI_Waitall` and `MPI_Wait` operations dominate the P2P communication time. The use of non-blocking operations indicates potential communication and computation overlap.



(a) Percentage of P2P, collective and computation time.

(b) Time spent in communication (P2P + collective) and computation.

Figure 3.5: Total time spent in communication (point-to-point and collective) and computation by the middle rank process. We observe that majority of applications spent more time in communication and communication time is dominated by collective operations.



(a) Illustration of MPI point-to-point function usage. We observe that non-blocking send and receive are the most used functionality.

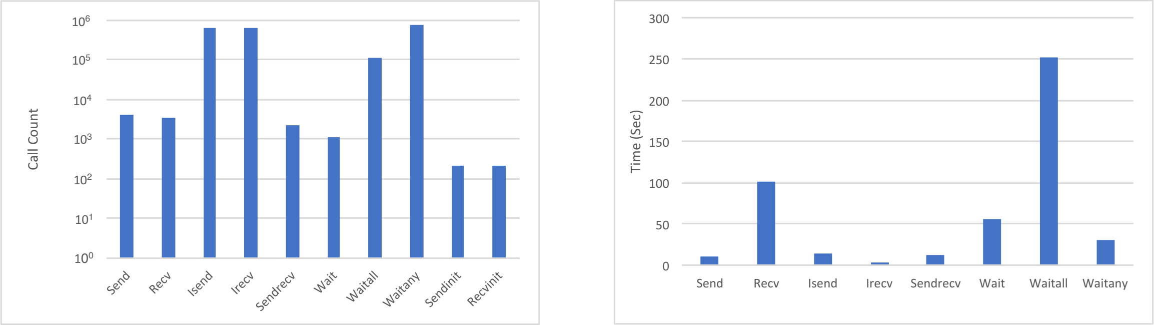
(b) Illustration of MPI point-to-point function usage along with potential communication and computation overlap.

Figure 3.6: Overview of MPI point-to-point communication for the rank 0 process.

`MPI_Send_init` and `MPI_Recv_init` time tends to zero, so we do not include it in the figure.

MPI point-to-point operations (receive and probe) allow the use of *wildcard* for source process rank and/or tag. It indicates that a process will accept a message from any source and/or tag. The source wildcard is `MPI_ANY_SOURCE` while the tag wildcard is `MPI_ANY_TAG`. The scope of these wildcards is limited to the processes within the specified communicator.

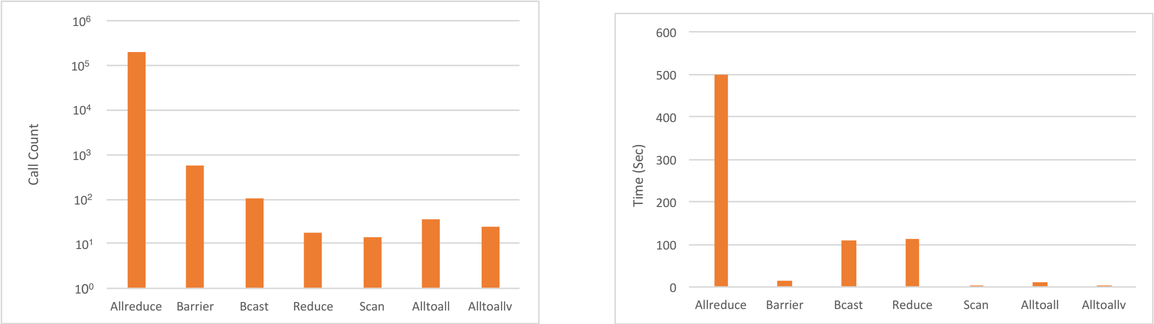
A small number of applications (four) use a wildcard in receive and probe operations. Three applications, AMG, MACSio, and NEKbone use the MPI source wildcard and only one application, PICSARlite uses the MPI tag wildcard.



(a) Illustration of MPI point-to-point function usage. We observe that nonblocking send and receive are the most used functionality.

(b) Illustration of MPI point-to-point function usage along with potential communication and computation overlap.

Figure 3.7: Overview of MPI point-to-point (P2P) communication for the middle rank process.



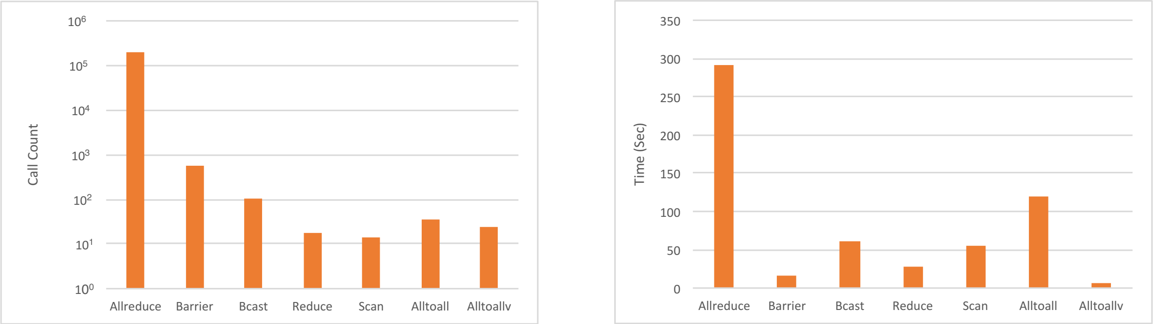
(a) Illustration of MPI collective function usage. We observe that Allreduce is the most used functionality.

(b) Illustration of MPI collective function usage. We observe that Allreduce dominates the communication time.

Figure 3.8: Overview of MPI collective communication for the rank 0 process.

MPI Collective Operations. The aggregated count of collective operations for the processes of rank 0 and the middle rank is shown in Figure 3.8a and 3.9a, respectively. Figure 3.8b and 3.9b show the time spent in the processes of rank 0 and the middle rank for each collective call. The figure does not include `MPI_Gather` because SW4lite calls it only once. In terms of call count and time, `MPI_Allreduce` is the most significant collective operation. Our results show that the rank 0 process spent more time in `MPI_Allreduce`, `MPI_Barrier`, `MPI_Bcast`, and `MPI_Reduce` while the middle rank process spent more time in `MPI_Scan`, `MPI_Alltoall`, and `MPI_Alltoallv`.

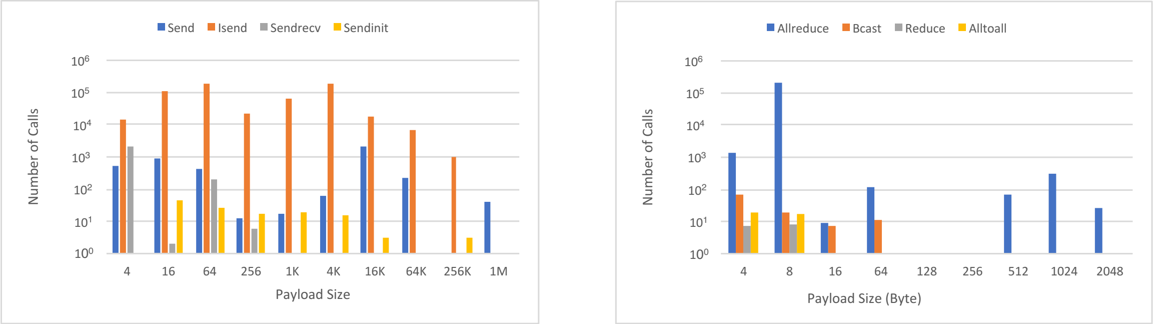
Payload Distribution. Figure 3.10a shows the distribution of payload size of point-to-point (P2P) send operations (`MPI_Send`, `MPI_Isend`, `MPI_Sendrecv`, and `MPI_Sendinit`) for



(a) Illustration of MPI collective function usage. We observe that Allreduce is the most used functionality.

(b) Illustration of MPI collective function usage. We observe that Allreduce dominates the communication time.

Figure 3.9: Overview of MPI collective communication for the middle rank process.



(a) Illustration of point-to-point payload distribution. We observe that P2P operations use a wide range of messages (from small to large).

(b) Illustration of collective payload distribution. We observe that majority of collectives use small messages.

Figure 3.10: Overview of payload distribution by the rank 0 process.

the rank 0 process. Our results show that more than 60% of the send uses a message length of less than or equal 1KiB.

The payload distribution for collective operations is shown in Figure 3.10b. All of the collectives use very small payloads. The figure shows that around 90% of the collective messages are less than 128 bytes. We found some instances of `MPI_Allreduce` that use a payload greater than 1KiB.

3.5.3 MPI Datatypes

A compliant MPI implementation provides a number of predefined basic datatypes (e.g., `MPI_INT`, `MPI_FLOAT`, `MPI_DOUBLE`), corresponding to the primitive data types of the programming languages. An MPI implementation allows user to construct derived types from existing types (basic and derived). There are several methods to construct derived types.

- **Contiguous.** Creates a new contiguous datatype by concatenating user defined `count` copies of an existing type.
- **Vector.** Creates a single datatype representing elements separated by a constant distance (stride) in memory.
- **Indexed.** Defines a new datatype consisting of an user defined number of blocks of arbitrary size. Each block can contain different number of elements.
- **Struct.** Creates a structured datatype from a general set of datatypes.
- **Subarray.** Creates a new MPI type describing a n-dimensional subarray of a n-dimensional array. The subarray may be situated anywhere within the array.

Table 3.4 presents the derived types used in the proxy applications. Most of the MPI operations use predefined basic types for communication. However, we found six applications in the suite that use derived types. But, only a few instances of the communication calls use the derived types. The majority of the derived datatypes are used in point-to-point communications. Only two applications (MACSio and miniVite) pass user-defined "struct type" in collective calls. Apart from these six applications, the libraries used by Laghos and thornado-mini uses derived datatypes in a small number of point-to-point communication.

3.5.4 Communicators, Groups, and Topology

An MPI implementation typically supports two types of communicator.

Table 3.4: Usage of MPI derived datatypes

Application	Contiguous	Struct	Subarray	Vector
AMG		×		
MACSio	×	×		
miniVite		×		
PICSARlite	×	×	×	×
SW4lite				×
SWFFT	×		×	

- **Intra-Communicator.** It is used for communication within a single group of processes. For collective communication, it specifies the set of processes that participate in the collective operation.

- **Inter-Communicator.** It is used for communication between two disjoint groups of processes. The group containing a process that initiates an inter-communication operation is called the "local group". The group containing the target process is called the "remote group".

The default communicator for MPI is `MPI_COMM_WORLD` (an intra-communicator), which is available after MPI initialization. It consists of all the processes in an MPI job. However, an MPI application can create new communicators to communicate with a subset of processes. Also, the MPI Standard, MPI-3.0, introduces *Neighborhood collectives* to enable communication on a process topology. It allows the user to define their own collective communication patterns. All processes in the communicator are required to call the collective. It has both blocking and non-blocking variants.

Our analysis finds that the majority of the proxy applications only use the default communicator, `MPI_COMM_WORLD`. Two of the applications, NEKbone and MACSio, use a duplicate of `MPI_COMM_WORLD`. Table 3.5 presents the applications that create new communicator along with the MPI routines used for the creation. We identified five applications that create new communicators by using `MPI_Comm_create` and `MPI_Comm_split`. Three of these applications use Cartesian topology to order the MPI processes. Although three applications use process topology; they do not use any neighborhood collectives operations. Our analysis also shows that the library "hypre" creates its own communicators (not duplicate of `MPI_COMM_WORLD`).

Table 3.5: Usage of MPI communicator, group, and topology

Application	Communicator and Group	Process Topology
AMG, Laghos	MPI_Comm_create, MPI_Comm_split MPI_Group_incl	
miniAMR	MPI_Comm_split	
PICSARlite		MPI_Cart_create
SW4lite	MPI_Comm_create, MPI_Comm_split MPI_Group_incl	MPI_Cart_create
SWFFT		MPI_Cart_create, MPI_Cart_sub

The proxy application "Laghos" uses "hypre" and thus indirectly uses the functionality. None of the applications from the proxy apps suite create inter-communicators.

3.5.5 Dynamic Process Management

To create a new process in an already running program, MPI uses `MPI_Comm_spawn` routine. This routine returns an inter-communicator.

Our analysis shows that none of the applications use the process management feature even though it was introduced long ago (in version 2.0, 1997). We hypothesize that the reason is that most production environments use batch schedulers, and such schedulers a) have little or no support for dynamic changes in the runtime size of MPI programs, b) production applications specify their maximum size of process utilization at startup, and c), most programs think in terms of the build-down from `MPI_COMM_WORLD`, rather than a build-up mode of process creation and aggregation. The latter is also complex-to-cumbersome syntactically (e.g., merging multiple intercommunicators).

3.5.6 MPI I/O (File)

The MPI Standard provides support for parallel I/O using its own file handle `MPI_File`. In MPI, the file open and close operations are collective over a communicator. However, the communicator will be usable for all MPI routines (other point-to-point, collective, one-sided, and unrelated I/O); the use of the communicator will not interfere with I/O correctness.

Only one of the proxy applications (miniVite) uses the MPI I/O functionality. However, it only uses MPI I/O to read binary data from a file, which is optional for this application execution. MACSio, a multi-purpose scalable I/O proxy application, uses Silo I/O library. None of the other proxy applications in the ECP suite use MPI I/O.

3.5.7 Error Handling

The set of errors that can be handled by MPI is implementation dependent. There are two predefined error handlers provided by MPI.

- **Error Fatal.** MPI_ERRORS_ARE_FATAL is the default error handler. It causes the program to abort on all executing processes.
- **Error Return.** MPI_ERRORS_RETURN returns an error code to the user. Almost all MPI calls return a code that indicates successful completion of the operation. If an MPI call is not successful, an application can check the return code of that call and executes a suitable recovery model[4].

Our static analysis find that only two applications (MACSio and miniAMR) set error handlers for `MPI_COMM_WORLD`. MACSio uses `MPI_ERRORS_RETURN` to check the return code of MPI calls while miniAMR uses `MPI_ERRORS_ARE_FATAL` error handler.

3.5.8 MPI Tools and the Profiling interface

The MPI standard, MPI 3.0, introduces the MPI Tools interface (`MPI_T`) to access internal MPI library information. Since its inception, the MPI standard also provides the MPI profiling interface (`PMPI`) to intercept MPI calls and to wrap the actual execution of MPI routines with profiling code. None of the proxy applications of Table 3.1 use either of these interfaces of the MPI Standard.

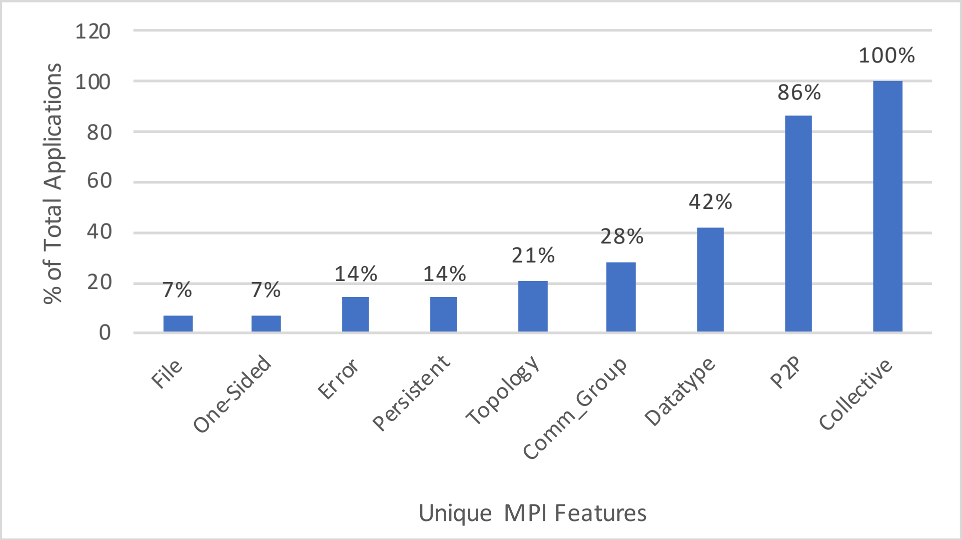


Figure 3.11: Illustration of MPI features usage by percentage of applications. We observe that point-to-point and collective features are most often used.

3.5.9 Overall use of MPI

In this section, we present the overall usage of MPI features in the proxy applications. Figure 3.11 presents the MPI usage statistics of the ECP proxy apps suite. Our results show that all of the applications use collective communication routines. While the percentage of applications using P2P communication is above 85%, less than 15% use persistent point-to-point and one-sided communication. Blocking and non-blocking point-to-point calls are more prominently used than either persistent point-to-point or one-sided calls. None of the applications use the process management or neighborhood collectives features.

Figure 3.12 shows the per-application usage of MPI features. Different applications use different sets of MPI features. None of our applications uses all 11 features. We observe that two applications use only collectives for communication. Our analysis also shows that almost 30% of the applications use only point-to-point and collective features from the standard. None of the applications use more than five MPI features. The proxy applications only use a small set of features from the MPI Standard.

[4] MPI-4.0, the next release of the MPI Standard, makes significant improvements to the situations where continuation after errors is possible. We considered applications based on MPI-3.1, where such continuation is iffy at best post error.

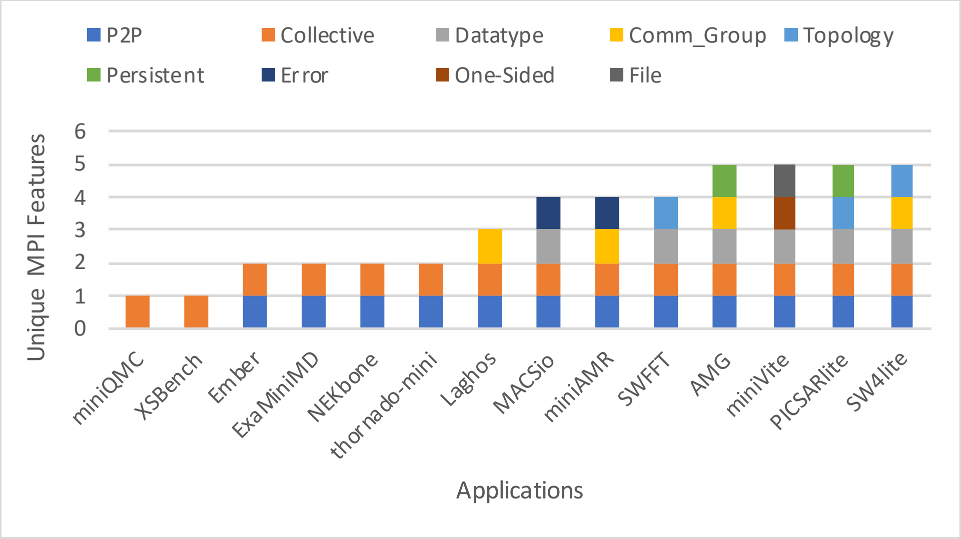


Figure 3.12: Illustration of usage of unique MPI features by applications. We observe that the majority of applications use a small fraction of the MPI functionality.

3.6 Conclusions

We empirically analyzed 14 ECP proxy applications to understand their MPI usage patterns. We used the MPI profiling interface (PMPI) to collect MPI usage statistics of the applications. Our study showed that these applications each use a small subset of MPI, with significant overlap of those subsets. Almost 50% of the applications use a hybrid programming model (i.e., MPI+OpenMP in most cases); however, the MPI threading environment is not used. The MPI collectives dominate the communication compared to point-to-point operations. The blocking and non-blocking point-to-point operations are more widely used than persistent or one-sided routines—a surprising finding given that the latter could potentially improve performance over the former.

Since these proxy applications use only a small subset of MPI and they represent the key characteristics of real exascale applications, it may therefore be worth focusing optimization and tuning efforts on implementing a subset of the MPI functionality, particularly in view of the large size of the overall MPI function set. The opportunity to focus optimization, hardware-offloading, and fault-tolerance on such subsets could well yield higher performing MPI libraries and faster time to solution for the real applications that these 14 ECP proxy applications represent.

Chapter 4

ExaMPI: A Modern Design and Implementation to Accelerate Message Passing Interface Innovation

4.1 Introduction

MPI is the standard specification for message passing libraries. There are multiple open source production-quality implementation (e.g., OpenMPI [17], MPICH [16]) of MPI Standard available , and many high-end vendors base their commercial versions on these source bases (e.g., Intel MPI [20], Cray MPI [19], IBM Spectrum MPI [21], MVAPICH [22]). All of these currently available MPI libraries are monoliths (implements the full MPI specification). The complexity of these leading open source implementations of MPI is daunting when it comes to experimentation and modification with new and different concepts for MPI-4 or other research experiments. This makes deep experimentation with or changes to MPI prohibitive, except in device drivers and incremental APIs. For instance, changing the modes of progress or the modes of completion of MPI implementations is a tall order, as is managing their ability to cope with internal concurrency or state. Furthermore, constrained environments, such as embedded devices and FPGAs, may also prefer to execute MPI functionality without coping with the entirety of large middleware implementations.

Production open source MPIs have successfully focused on completeness of coverage, correctness, compliance, and, of course, middleware portability and performance. But, they typically leveraged software architectures rooted in legacy implementations of MPI-1 or earlier message passing systems, where assumptions were made based on then-extant architectures, processor resources, assumptions of intra-node concurrency, and performance levels. Production open source MPIs possess complex internal architectures, layers, and global state, and cross-cutting issues can arise when trying to experiment. Such issues make it difficult and expensive to achieve new changes, while also limiting certain kinds of experiments like overlapping of communication and computation.

To enhance and simplify researchers' ability to explore new and diverse functionality with MPI with quality performance potential, we have devised ExaMPI, a new, BSD-licensed open source implementation. ExaMPI is designed as a research vehicle for experimenting with new features or ideas of MPI. There are several factors that motivates the design of ExaMPI. First, to explore new fault-tolerant models for MPI. Second, the recognition that many MPI applications require only a small to moderate subset of functionality also motivated our design of a new research MPI implementation. Most MPI applications use a small set of functionality from MPI Standard [12, 11], which means the complexity associated with full API support isn't needed for many kinds of applications and, hence, application experiments. By supporting a sound design and allowing functionality to be added systematically over time, we provide incremental ability to run practical codes while reducing the total amount of MPI middleware by orders of magnitude. Furthermore, with a sound, first-principles design, this new MPI would have little dead code, or the technical debt associated with assumptions about node concurrency or progress made in the 1990s. Third, over the past few years, a number of robust data movers such as Libfabric, Portals, and even InfiniBand verbs have decreased the importance of complex "channel devices" and other transport abstractions within MPI itself. These transports often include internal progress (independent of user calls to MPI) for sufficiently smart NICs and will soon include collective communication offload for some networks. This growth has enabled ExaMPI's design to focus on Libfabric as the key production networking interface for ExaMPI in addition to fundamental UDP/IP and TCP/IP network drivers. Also, ExaMPI allows us to avoid legacy issues in the code base of existing mature implementations as well as enforce modular design goals throughout all functionality.

The intention is *not* to replace key blocks of functionality or policy in existing production MPI implementations such as MPICH and Open MPI, or in commercial derivatives thereof. Rather, we intend to: (a) enable rapid prototyping of new algorithms or operations for MPI, (b) identify opportunities to improve MPI at-large, (c) exploration of new fault-tolerant models for MPI, and (d) integration of multiple fault-tolerant MPI models for more fine-grained failure recovery.

The key contributions of this work are as follows:

(1) Introduces a simplified and modular MPI implementation for research purpose with no legacy issues.

(2) Open source (BSD-licensed)implementation that enables a community of contributors.

4.2 Requirements

We gathered both functional and non-functional requirements for ExaMPI development. In certain cases, these requirements are related to the modularity and extensibility of the software itself for use in specific applications.

The following functional requirements were identified at the outset of the project:

(1) Support most commonly used subset of the MPI 3.1 standard instead of implementing full MPI specification

(2) Achieve point-to-point throughput that is initially competitive with production-quality open source MPI implementation (e.g., OpenMPI)

(3) Achieve latency that is appropriate for a strong progress implementation

(4) Implement the **MPI Stages** model of MPI fault tolerance (detail discussion in Chapter 5)

Followings are the non-functional requirements:

(1) Use C++ as the programming language in the development

(2) Design a modular and extensible library that is easily maintainable

(3) Enable strong progress (independent progress)

(4) Enable choice of polling and blocking notification

(5) Support socket based transport (e.g., UDP, TCP)

(6) Enable experimentation with new fault-tolerance models for MPI

(7) Focus on persistent and non-blocking operations as fundamental, rather than blocking (point-to-point and collective)

(8) Enable a community of contributors of compatible extensions that are BSD-license compatible

4.3 Design

In this section, we provide an overview of the architecture of ExaMPI. The library is designed to accommodate modules with well-defined interfaces. Here, we discuss different modules of the ExaMPI library (e.g., progress, transport, interface).

4.3.1 Architecture of ExaMPI

Figure 4.1 presents the Unified Modeling Language (UML) class diagram of the ExaMPI library. This diagram contemplates both the top-down view of the standard APIs and data structures and the bottom-up view of data movers implementing transports for MPI. Each class has well-defined interfaces that encapsulate its behavior. The encapsulation enables cohesive classes of loose coupling, which makes the library easily understandable and maintainable.

4.3.2 Progress Engine Design

In MPI, ‘Progress’ is the software stack that resides above the network layer responsible for the progression of outstanding messages. The Progress rule of MPI Standard has two different standard compliant interpretation—(a) once a communication operation has been posted, the subsequent posting of a matching operation will allow the original one to make progress regardless of whether the application makes any further library calls (independent message progress) and (b) application (on the receiver/target process) requires to make further library calls to make progress on other pending communication operations (polling). Also, an MPI implementation can use polling and interrupt (blocking) for message completion notification. Figure 4.2 shows

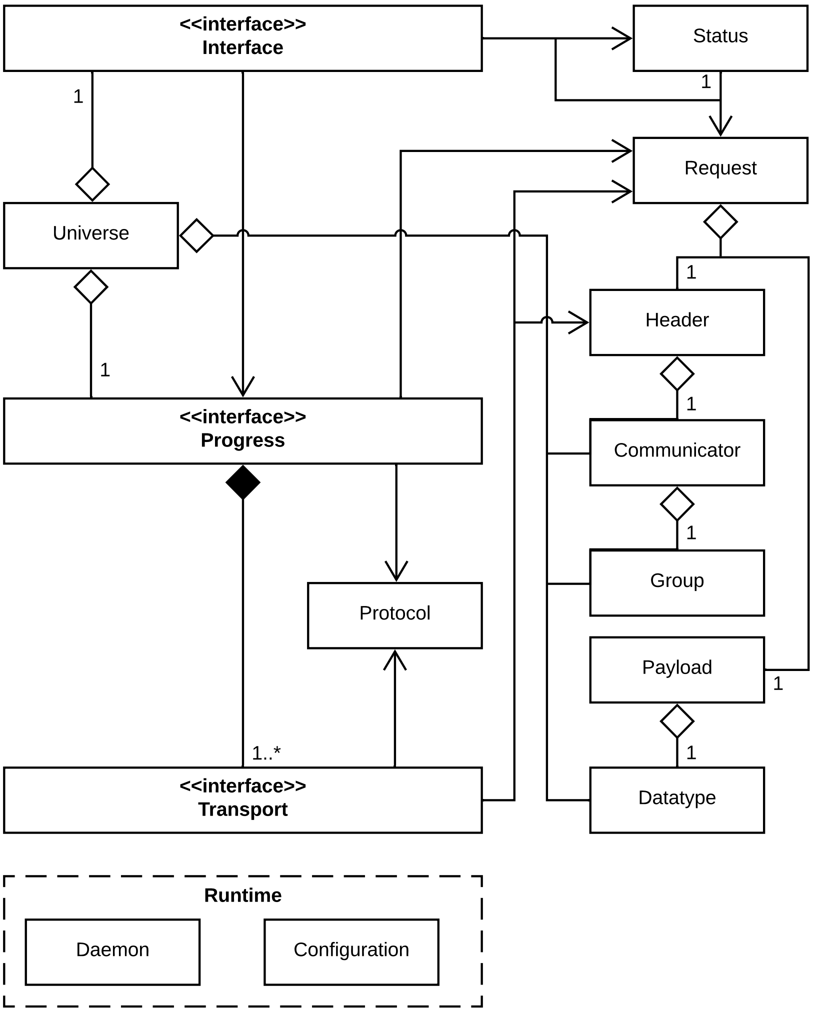
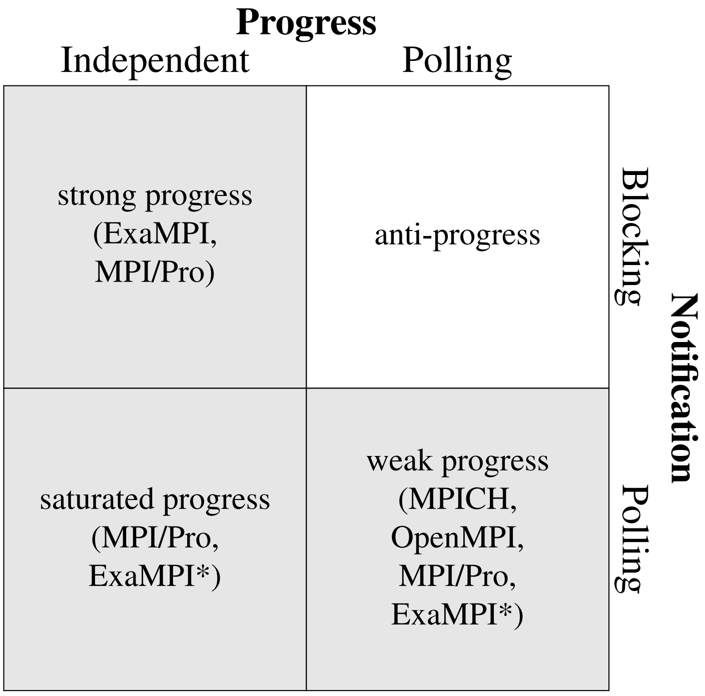


Figure 4.1: General overview of the ExaMPI library.

different classes of MPI message progress and notification engines that are possible to be constructed [81]. In ExaMPI, we enable all these possible combinations of message progress and notification. The progress engine abstraction is designed to allow any progress engine to be implemented inside the library rather than having a fixed one. This feature is important because prior work has shown that overlapping of communication and computation is severely hampered by polling behavior in progress and/or notification [82].

We restrict all progress to be made through the progress engine by requiring all operations to construct a request object. The request object is posted to the progress engine, which then will progress the request objects and the underlying transport implementations. Currently,



Notification \ Progress	Independent	Polling
Blocking	strong progress (ExaMPI, MPI/Pro)	anti-progress
Polling	saturated progress (MPI/Pro, ExaMPI*)	weak progress (MPICH, OpenMPI, MPI/Pro, ExaMPI*)

Figure 4.2: Overview of Dimitrov's Progress and Notification Classification Diagram (* Forthcoming modes in ExaMPI).

ExaMPI implements a strong progress engine. By "strong" we mean that the progress is independent with separate progress threads from the user threads and the notification of completion is blocking. When a user thread waits on a request, the user thread is unscheduled until the request is complete. Further progress engines are being developed to implement the weak and saturated progress classes.

Figure 4.3 shows the decomposition of the functionality within the Progress class. We separate the matching engine from the progress engine through an interface that allows us to implement many separate algorithms to perform matching. Currently the SimpleMatcher implements a unmatched message queue and posted received queue with a complexity of $O(N^2)$.

Message Matching. The matching condition checks for communication context, source, destination, and message tag. The context identifies a group of processes that can communicate with each other. Within a context, each process has a unique identification called rank that is used to specify the source and destination process for the message. Finally, there is a user defined tag value that can be used to further distinguish messages. A receive call matches a message when the tag, source rank and context specified by the call matches the corresponding values.

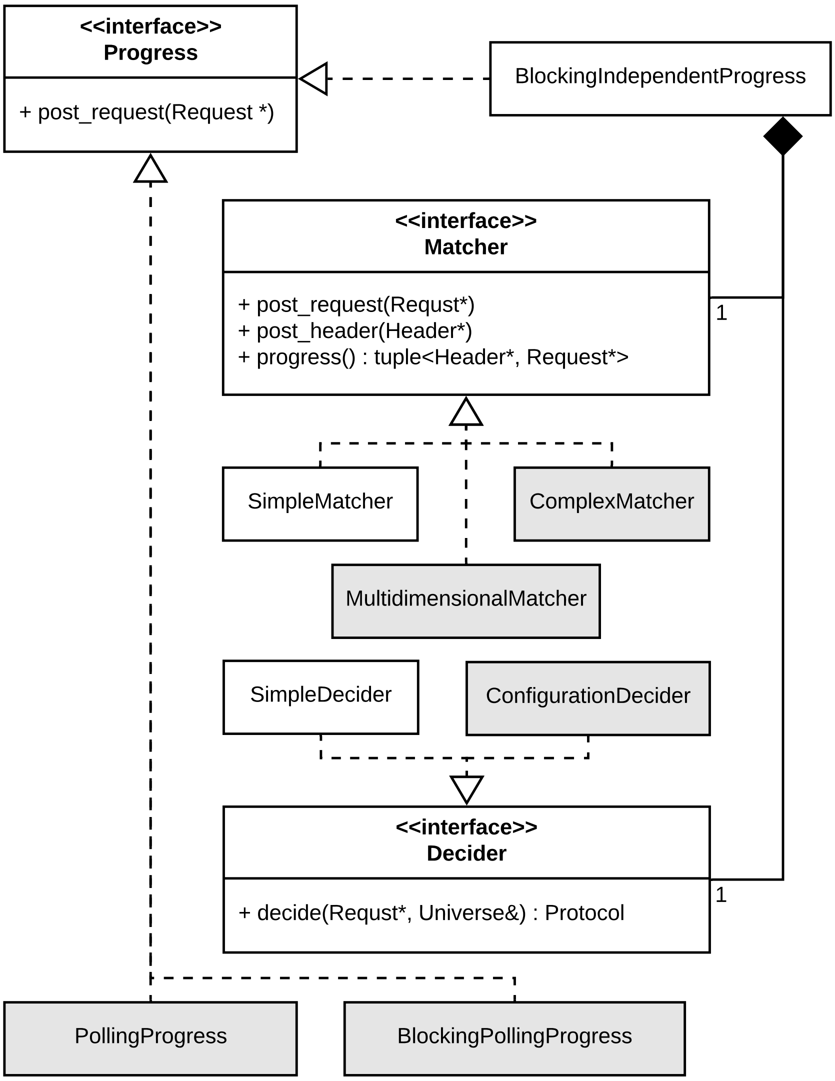


Figure 4.3: Overview of the Progress layer in ExaMPI.

In addition, we decompose further the mechanism for decision about which protocol (or algorithm) is to be used for any MPI operation. The SimpleDecider object implements the expected behaviour of the point-to-point functionality. By implementing a custom decider class, developers can map any MPI operation to any underlying protocol. Every MPI implementation generally relies on at least two protocols:

Eager Protocol. The sender assumes that the receiver has enough memory space to store the message; this allows to send the entire message immediately, without any agreement between peers. It has minimal startup overhead and provides low latency for small messages.

Rendezvous Protocol. Some kind of “handshaking” between the processes involved in the transfer must be implemented. It allows to transfer big messages with minimal impact on performance.

Currently, ExaMPI only uses the Eager protocol to send point-to-point messages.

4.3.3 Transport Design

The transport layer present within ExaMPI is intended to allow abstraction of all available network APIs. Figure 4.4 shows the hierarchy and required functions any Transport class currently is required to have implemented. Further development on this aspect of the library will enable shared memory transport, offloading collectives and one-sided remote memory operations.

The current implementations present are the UDP and TCP transports, which allow for global usage but are not as performant as a high-performance network. Each transport implementation is entirely responsible for handling the memory associated with the network. As such, TCP and UDP use the kernel IP stack as a form of network buffer, but buffer payloads separately once received. Future implementations with more complex communications fabrics will require handling of receive queues.

4.3.4 Interface

Here, we overview the interface layers (language bindings) of ExaMPI library. The MPI Standard defines bindings for both C and Fortran, which must be available from any compliant MPI implementation. Currently, ExaMPI only provides the C language interface, but building the Fortran interface is trivial above the current implementation, similar to other MPI implementations that build their Fortran bindings to simply call the C bindings.

In Figure 4.5, the current interface structure of ExaMPI is presented. The C symbol names for both the MPI layer and PMPI layer are defined in the *mpi.h* header file. The MPI symbols are defined to be weak linked to facilitate the overloading of their functionality by MPI-compatible tools. The default MPI functions directly call the equivalent PMPI function.

The Interface class declares and defines the same interface as the C bindings of MPI but within a C++ class structure. The PMPI layer uses the Universe object to find the interface to the

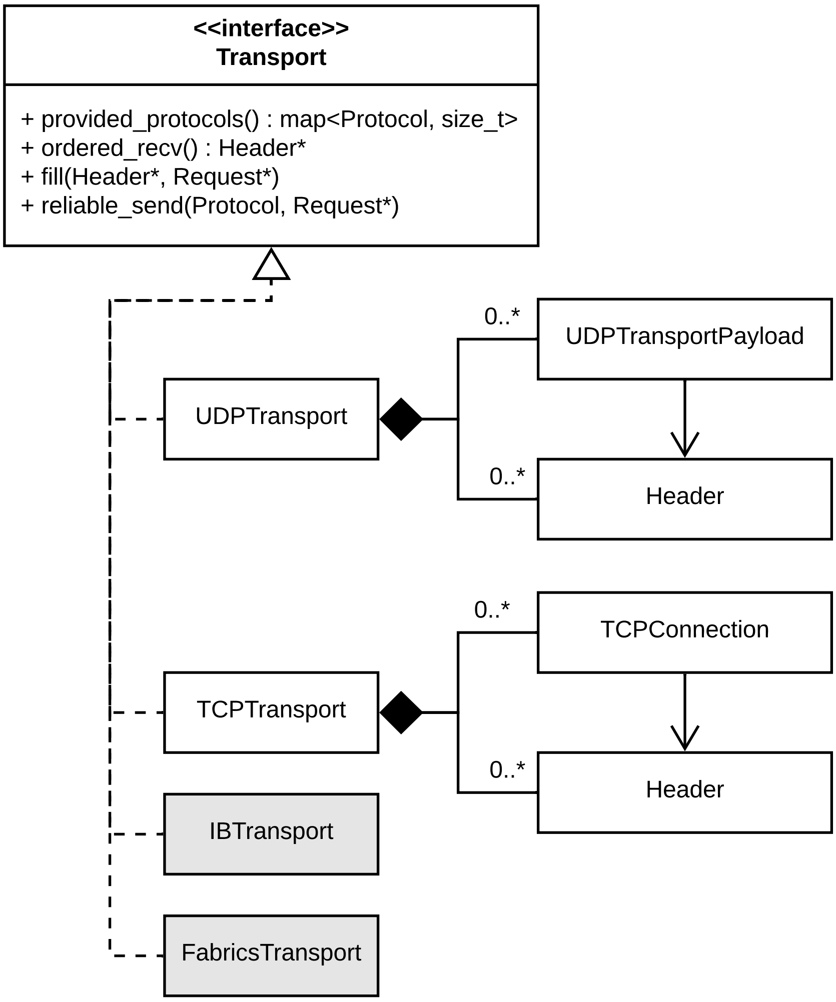


Figure 4.4: Overview of the Transport layer in ExaMPI.

underlying C++ interface. This structure allows the abstraction of various interfaces for further work such as additional functionality for MPI Stages recovery model (StagesBasicInterface).

In addition to extendability, the BasicInterface class allows us to encapsulate all top-level MPI behavior into a single location, which includes error checking and subsetting of blocking and non-blocking paths into persistent path, which is implemented by the underlying library.

Currently, the ExaMPI library supports a subset of MPI-3.1 functionality. Appendix A presents the list of currently implemented MPI functions. Here, we do not include the list of MPI API extensions implemented to support MPI Stages fault model (discussed in Chapter 5).

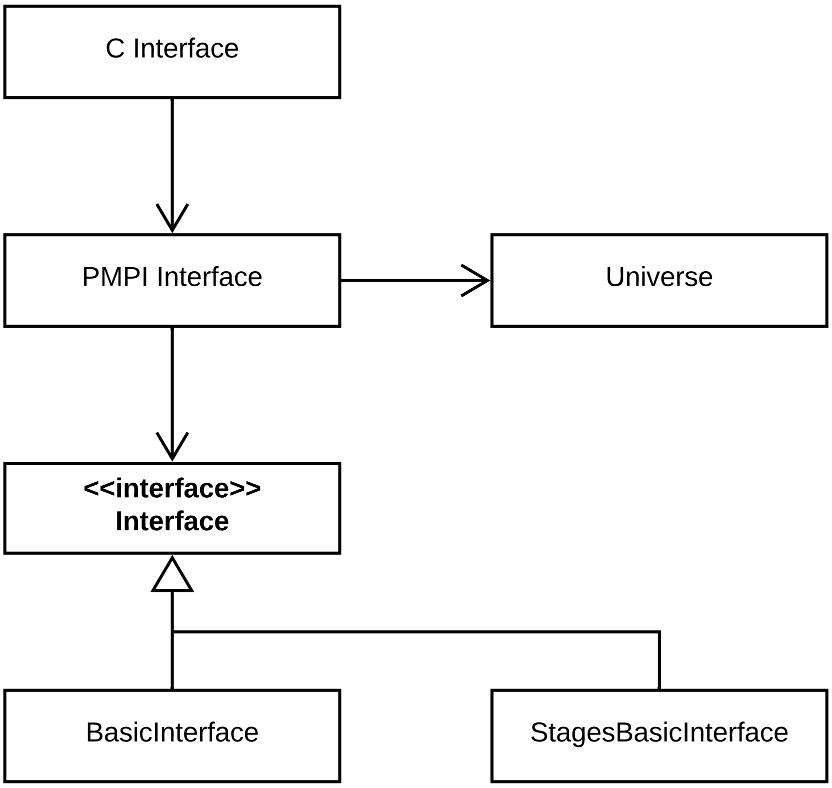


Figure 4.5: UML diagram describing interface layers of ExaMPI.

4.3.5 Universe

In ExaMPI, the Universe is a special class to avoid global state. It is implemented as a singleton class. The Universe owns all global states of the library (e.g., request objects, communicators, groups, datatypes). Currently, we do not support MPI derived datatypes. Our implementation only uses pre-defined MPI datatypes (e.g., `MPI_INT, MPI_DOUBLE, MPI_FLOAT_INT,` etc).

4.3.6 Runtime

The MPI runtime system launches an MPI application. ExaMPI uses `mpiexec` command to initiate the parallel job. `mpiexec` command takes multiple arguments. For instance, $mpiexec -n\ X\ [args] < program >$ will run X copies of $program$ in the current runtime environment. ExaMPI's mpiexec use SLURM [83] resource manager as opposed to ssh, which requires the use of a hostfile. The runtime environment of ExaMPI also supports execution of MPI application in localhost.

4.3.7 Utilization of C++17

The ExaMPI implementation is written using C++17, which enables many productivity and language features that are not present in earlier C++ or C specifications. In addition, C++ allows for object-oriented programming, which allows the MPI implementation to directly deal with objects instead of the handles to objects. Using objects allows us to develop expressive source code without the clutter required with a C implementation.

We intend for ExaMPI to support full thread safety—with as much internal concurrency as reasonably possible—through the entire library. This arrangement is currently achieved with locks provided by the C++11 specification. By utilizing the built-in threading facilities, we reduce our dependence on external libraries. In the future, we will develop thread-safe, lockless data structures that will allow for the overhead of locking to be removed. They are also supported with built-in atomic operations.

The C++ language also provides many capabilities that are tedious to use in C. One of these capabilities is string objects, which provide simpler handling of textual data. Another is exception handling. In C, the error code mechanism requires branches through the code base and forces design decisions. Within C++, exceptions are provided and allow for much cleaner internal working with errors. We utilize exceptions throughout the internal MPI library but provide error codes to the top-level MPI layer.

4.4 Conclusions

In this chapter, we described ExaMPI, a new, experimental implementation of the MPI Standard. ExaMPI solves the problem that full-scale open source MPIs are legacy middleware projects of large-scale and long-running development by many contributors; they are difficult to learn, modify, and use for middleware research, except in limited ways. Where they are usable, they are adequate, but many experiments are either intractable or require students and professors to spend inordinate amounts of time "modifying around the edges" of such middleware.

Thus far, ExaMPI has proven to be a useful research vehicle for a small number of people. As we move to a community of developers, researchers, and users, we look to increasing that utilization dramatically and expect the modularity of design to allow for many interesting hybridizations of our baseline code and concepts with others' ideas, prototypes, and additions.

Chapter 5

Failure Recovery for Bulk Synchronous Applications with MPI Stages

5.1 Introduction

With each successive generation of large-scale parallel systems, faults and associated failures are becoming more frequent. Based on current knowledge of existing high-performance computing (HPC) systems, it is anticipated that failure rates may increase substantially in future exascale systems and MTBF will reduce to only few hours [3, 4]. Long-running scientific applications in such systems will require efficient fault-tolerance support [5]. The Message Passing Interface (MPI) is widely used for data communication in scalable scientific applications. The MPI Standard provides mechanisms for users to handle recoverable errors. However, the standard itself does not provide any mechanism to continue running MPI after a failure. Thus, there is a growing effort in the MPI community to incorporate fault-tolerance constructs into MPI. Several approaches have been proposed for MPI to tolerate failures (see Section 2.4.3.3). However, none of these has yet been integrated into the MPI Standard, a crucial precondition for widespread support and continued program portability.

According to the current standard, MPI-3.1, by default MPI programs automatically abort if a process fails [84]. For large-scale bulk synchronous applications, since a failure in a process quickly propagates to other processes [85], these applications typically handle failures by periodic, global synchronous checkpoint/restart (CPR). When a failure occurs, the runtime environment kills the entire job and restarts the application from the latest (or penultimate) global checkpoint.

While the CPR approach fits the MPI standard, the full job restart time in standard CPR is significant and ejection from the job queue is not an attractive alternative in production settings either. Also, in forthcoming exascale systems, killing the entire job for a single failure may be infeasible. Recently, the global-restart model was introduced [73, 86] to address these issues. In this model, the state of MPI is globally reinitialized and cleaned up *automatically* upon a failure, allowing both live processes (i.e., processes that survived the failure) and new processes (i.e., processes that replace a failed process) to be restarted from the beginning of the program without terminating the job. This method enables a number of optimizations that can significantly reduce failure recovery time, including avoiding job/resource allocation time, enabling the reuse of existing process connections, and enabling the application to load checkpoints from memory for live processes.

While the global-restart method is useful in reducing recovery time, *all* processes—whether they are live or new processes (replacing failed ones)—must jump back to the beginning of the program, That is, they must re-enter `main()`. Ideally, if a failure occurs in the main computation loop where the program spends most of its time, we would want only new processes to start from the program beginning and live processes to remain in the main computation loop, close to where the failure occurred. This would eliminate the need for the live processes to perform unnecessary initialization steps after entering `main()` and before entering the main loop. This form of recovery, however, is not possible with today's global-restart approaches and requires advanced MPI functionality to allow new processes to *reintegrate* with live processes waiting for them later in the computation (i.e., the main loop).

In this chapter, we introduce our work `MPI Stages` [9], a new global-restart model to support fault tolerance in bulk synchronous MPI applications that addresses the aforementioned issues of the global-restart model. Each *stage* is a period between synchronous checkpoints and provides a temporal containment of faults; stages are numbered by a user-visible *epoch* maintained by MPI and the underlying runtime system. In this model, a checkpoint of the MPI state is saved along with the application state: this MPI state can contain state that is visible to the user (e.g., MPI communicators), and MPI state that is only visible to the MPI library (such as network connectivity and queues).

MPI Stages allows the runtime system to replace a failed process transparently by restoring its MPI state and application state prior to the failure. Live processes keep the MPI state they have before the failure, which allow them to stay in the main computation loop while the replacement process reintegrates with them. Our approach targets bulk synchronous parallel computing applications, a model that is used in the vast majority of large-scale HPC applications. A fundamental challenge with this model is that we must be able to quantify and capture MPI state during normal operation in a meaningful and complete way.

We introduce new API calls for MPI to support state checkpoint: `MPIX_Checkpoint_read()`[1] to read the saved MPI state for a process, `MPIX_Checkpoint_write()` to save the current MPI state of a process, `MPIX_Get_fault_epoch()` to retrieve which checkpoint (last synchronous) to load. We also introduce functionality in MPI to revivify MPI handles after failure. We add an error code `MPIX_TRY_RELOAD` to let an application know when it should stop normal operation and move to retrieve the checkpoint (and continue execution thereafter).

In summary, our contributions are as follows:

(1) We introduce the concept of MPI Stages, a global-restart model to reduce the recovery time of stage-based bulk synchronous MPI applications.

(2) We identify requirements for production MPI implementations to support state checkpointing with MPI Stages, which includes capturing and managing internal MPI state, and serializing and deserializing user handles to MPI objects.

(3) We design a double-dæmon runtime system to transparently replace a failed process which is compatible with various job scheduling systems.

5.2 Terminology

In this section, we introduce the terminology used in the design of `MPI Stages`.

Stages— a period between synchronous checkpoints for both MPI and application states and provides a temporal containment of faults.

[1] 'X' indicates non-standard MPI API

epoch— this is the "stage" notion of our design. Each stage is numbered by an epoch. This epoch is not to be confused with MPI RMA epochs. "Epoch" could literally have been named "stage" had we so chosen. In this work, the epoch represents a "temporal context" as compared to the MPI-internal concept of "spatial contexts" used in MPI communicators to scope messages.

first-time start process— when an MPI program first begins, all processes are categorized as first-time start processes.

live process— upon a failure, the processes that have survived the failure are categorized as live processes.

relaunched process— during recovery from a failure, processes that replace the failed processes are categorized as relaunched processes.

resilient_loop— a loop that executes a resilient MPI program using `MPI Stages` recovery model. All of the application code resides inside the resilient_loop. If a process fails during the application execution, resilient_loop takes appropriate action to recover the failure. The resilient_loop exits when the application is successfully complete or encounters any failures not supported by `MPI Stages` recovery model. MPI processes call finalize once resilient_loop completes.

main_loop— a function that contains the main simulation of bulk synchronous program. `MPI Stages` separates the application into two functions— `main` and `main_loop`. The `main` function consists of a "resilient loop" that initializes MPI and calls the `main_loop`, which executes all of the application code. The `main_loop` successfully returns when the application is complete. In Algorithm 2, the "while" loop in the main function is the resilient_loop and it calls the main_loop function which contains the application code.

MPI state— an MPI state is defined as the current information pertaining to MPI processes, groups, and communicators. During program execution, the latest values of these identified MPI objects are checkpointed.

Algorithm 2 Structure of MPI application using MPI_Stages

```
procedure main(int argc, char **argv)                          ▷ main function
    while not $is\_success$ do                                 ▷ resilient_loop
        ...
        $is\_success = main\_loop(...)$                        ▷ call main computation
    MPI_Finalize()
procedure main_loop(...)                                       ▷ main_loop function
    all application code here
    returns success/failure
```

5.3 Fault Model

For the purpose of this dissertation, we only consider fail-stop *process failures* and detectable transient errors (e.g., unwanted bit-flips). The resource manager or MPI runtime is responsible for restarting any failed processes. We assume that processes can be re-spawned to replace the failed ones in the same node. Currently, we do not handle permanent hardware failures. However, the resource manager could be used to replace the failed processes with spares. As our recovery is conducted in a resilient_loop, software errors (bugs) could result an infinite recovery loop. In this case, the resilient_loop should abort after certain number of tries.

5.4 Applicability of MPI Stages

An MPI application can use different types of recovery strategy based on its programming model. In this section, we discuss different recovery models available to MPI applications, the recovery model for MPI Stages, and the programming model that best fits MPI Stages recovery model.

Shrinking Recovery discards the failed processes and the application recovers with a reduced set of processes. As a result, for many scientific problems, it requires the ability to re-decompose and load balance them at runtime.

Non-shrinking Recovery replaces the failed processes and the application recovers with same number of processes it had before failure. In this case, the resource manager may have to maintain a pool of spare processes; however, application need not do any load balance and re-decomposition.

Forward Recovery attempts to find a new state from which the application can continue execution. ULFM model uses a forward recovery approach.

Backward Recovery restarts the application from a previously saved state. Reinit model uses this approach.

Local Recovery restores a small part of the application to survive the failures. It assumes that the impact of the failures is limited to a local state.

Global Recovery restores the global state of the application to repair the failure. In this case, even if only one process fails, all processes need to restore their state.

We use a global, backward, and non-shrinking recovery model in `MPI Stages`. This type of recovery model is best suited for bulk synchronous programs (BSP). BSP is one of the commonly used programming models in large-scale HPC applications. In BSP, the state of the system advances by time steps. It uses static domain decomposition and each process stores only its assigned part of the time evolving state. Using a global, backward, and non-shrinking recovery eliminates any complex re-decomposition and load balancing of the problem. Some of the applications use a master-worker programming model. In master-worker model, master distributes computation to the workers and workers return the result to the master. There is almost no dependencies among the workers. If a worker fails, the master can sends the failed workload to the next available worker without affecting their computation. A forward, local, and shrinking recovery model is best suited for this type of dynamically balanced application.

We primarily target bulk synchronous programs (BSP) that use checkpoint/restart as their recovery model. Here, we assume that the application has already managed the user's data as part of the application checkpoint and we apply our method on top of it. Our method requires the checkpoints taken at a globally consistent state (no pending requests). In BSP, we reach in a globally consistent state after each time-step (or n time-steps) of the simulation. We can also apply `MPI Stages` in master-worker applications; however, a global, backward, and non-shrinking recovery is not desirable for master-slave programming model.

5.5 Design

Any application-level fault-tolerant model for MPI requires extensions to the MPI Standard (MPI API). In this section, we discuss the key design features of `MPI Stages`. We also discuss the MPI API extensions introduced to use our recovery model.

5.5.1 Interface

Our model introduces only a small number of functions to MPI. In this section, we present the new objects and functions introduced in our model, as presently conceived. These are minimal additions at present, and no "giant API" addition is expected even in further elaboration of this approach.

We divide the added MPI constructs (objects and functions) into two categories— (a) *Minimal constructs*: required for any applications to use `MPI Stages` and (b) *Serialization constructs*: required for applications that create user MPI handles (see Section 5.5.4.4).

5.5.1.1 MPI Objects

In this section, we present the new objects and error code introduced by `MPI Stages`.

For MPI objects, we only add an error code as a minimal construct. Listing 5.1 shows the minimal construct required in our model.

`MPIX_TRY_RELOAD`— Error code returned to an application from the MPI library indicating the need to initiate the failure recovery process (e.g., checkpoint load).

The list of serialization constructs is as follows (details in Section 5.5.4.4):

`MPIX_Handles`— Structure that represents the MPI handles used in user program. Currently, we only support communicator and group handles.

`MPIX_Serialize_handler`— Function handler type for MPI handles that serializes user handles of MPI object used in the application and libraries.

`MPIX_Deserialize_handler`— Function handler type that revivify user MPI handles during recovery from a deserialized `MPIX_Handles` object.

Listing 5.2 shows the constructs required to revivify user handles of MPI objects after a failure.

5.5.1.2 MPI Functions

In this section, we present the new functions introduced by `MPI Stages`. We add extended semantics to `MPI_Init` to support our recovery (see Section 5.5.4). Here, we only show the newly added functions.

We add three MPI routines as the minimal construct to use `MPI Stages` (see Listing 5.1).

`MPIX_Checkpoint_read`— Load MPI state from a checkpoint and wait in an implicit barrier for all processes to complete the step.

`MPIX_Checkpoint_write`— Store MPI state into a checkpoint.

`MPIX_Get_fault_epoch`— Returns the last successful checkpoint number.

Followings are functions required for serialization (details in Section 5.5.4.4) (see Listing 5.2):

`MPIX_Serialize_handles`— Serializes user handles of MPI objects into MPI checkpoint.

`MPIX_Deserialize_handles`— Deserializes (revivify) the user handles of MPI objects from MPI checkpoint.

`MPIX_Serialize_handler_register`— Pushes serialize handlers in stack. Application and libraries can register their handlers using this function.

`MPIX_Deserialize_handler_register`— Pushes Deserialize handlers in stack. Applications and libraries can register their handlers that revivify their MPI handles using this function.

5.5.2 Failure Detection

Failure detection is one of the first steps of fault tolerance. Our model requires a perfect fault detector that is strongly accurate and complete. A perfect fault detector does not report a process as failed until it actually fails and eventually all peers will be aware of the failure (see

```
/********* MPI error code *********/
enum Errors {
 ...
 MPIX_TRY_RELOAD,
 ...
};

/********* MPI functions *********/
// function to load MPI checkpoint
int MPIX_Checkpoint_read(void);
// function to store MPI checkpoint
int MPIX_Checkpoint_write(void);
// function to get epoch
int MPIX_Get_fault_epoch(int *epoch);
```

Listing 5.1: Minimal MPI API extension to use MPI Stages

Section 2.4.1). We have a failure detector in our MPI runtime system to detect process failures; the runtime system could also detect network failures in future. Thus, applications do not need to check for failures. When a failure is detected by MPI runtime, the detector propagates the failure information proactively among the remaining surviving processes (as opposed to a gossip-based diffusion of such information).

Runtime System. One of the major design aspects of MPI Stages is the transparent replacement of failed process by the MPI runtime system. We use Slurm as a resource manager to allocate and maintain the resources (processes/nodes) requested by the application. To support our fault tolerance model, we design the runtime as a double-dæmon method. The job scheduling system has the management dæmon on the node that executes a second layer dæmon, the *fault dæmon*. The fault dæmon launches two more dæmons, *head dæmon* and *controller dæmon*. These new dæmons handle the failure detection and propagation of MPI Stages. One benefit of introducing a level of dæmon is that we do need to modify Slurm itself. So, it is compatible with other job scheduling systems. Figure 5.1 shows the hierarchy of different dæmons in our runtime system.

Fault Dæmon wraps the execution of the MPI application. It launches the MPI application as a sub-process. It intercepts any MPI process failure (non 0 exits) and prevents the job scheduling system from terminating the job prematurely. The fault dæmon also prepares the final application environment variables. The fault dæmon waits for the process to terminate and

```c
/********* MPI objects and function pointers *********/
typedef struct {
    MPI_Comm *communicators;
    int comm_size;
    MPI_Group *groups;
    int group_size;
} MPIX_Handles;
// function pointer to serialize MPI handles
typedef void (*MPIX_Serialize_handler) (MPIX_Handles *handles);
// function pointer to revivify MPI handles
typedef void (*MPIX_Deserialize_handler) (MPIX_Handles handles);

/********* MPI functions *********/
// function to serialize handles
int MPIX_Serialize_handles();
// function to revivify handles
int MPIX_Deserialize_handles();
// function to register deserialize handler
int MPIX_Deserialize_handler_register(const MPIX_Deserialize_handler handler);
// function to register deserialize handler
int MPIX_Deserialize_handler_register(const MPIX_Deserialize_handler handler);
```

Listing 5.2: MPI API extension for serialization of user handles of MPI objects

thereby avoids using any processing power from the node. This feature is useful because there is a one-to-one application and fault dæmon relationship. Figure 5.2 shows the state diagram of the fault dæmon. The fault dæmon also initiates the head dæmon if SLURM_LOCALID[2] (localid) is 0.

Head Dæmon is executed on each node. It manages a single node communicating with both local fault dæmon and the applications. In addition, it communicates with the controller dæmon. It initiates the controller dæmon if global task id for the process is 0 (root rank). It implements a “barrier” inside a node— receives barrier request from application process and sends the node barrier confirmation to controller. It also receives the “termination” request from the fault dæmon and notify controller about the successful/unsuccessful exit of the process. Head dæmon also participates in reaching a consensus on the epoch value after failure (see section 5.5.4.5). Figure 5.3 shows the state diagram of head dæmon.

[2] Node local task ID for the process within a job.

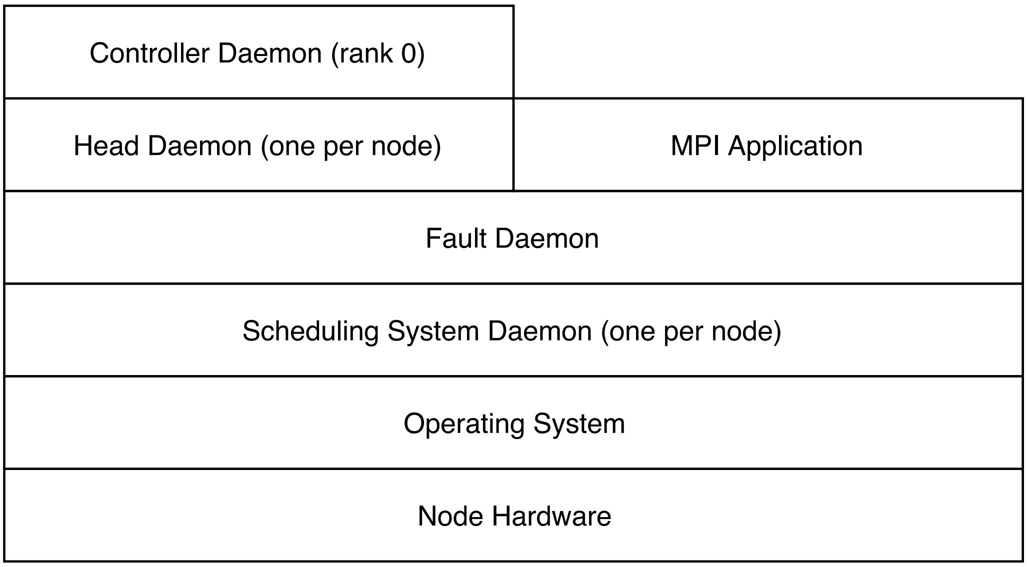

Controller Daemon (rank 0)
Head Daemon (one per node)
MPI Application
Fault Daemon
Scheduling System Daemon (one per node)
Operating System
Node Hardware

Figure 5.1: Overview of runtime system dæmon layers of `MPI Stages`.

Controller Dæmon is only awake on the root node (rank `0`) of the job allocation. It handles communication from all head dæmons. It listens to head dæmons and sends data back to them. It implements a “barrier” among all nodes— receives ‘node barrier’ request from all head dæmons and sends ‘release’ command back as a confirmation of the barrier. Similarly, it sends ‘shutdown’ commands to all head dæmons once all nodes exit successfully. The controller dæmon also propagates the failures among all head dæmons by sending ‘error’ command (see section 5.5.3). It also generates agreement on epoch value after a failure (see section 5.5.4.5). Figure 5.4 shows the state diagram of controller dæmon.

With this structure, there are two dæmons (head dæmon and controller dæmon) awake on the root node of the job allocation and one dæmon (head dæmon) on every other node. This arrangement enables hierarchical scaling for side-channel communication—that is, data not directly related to the MPI application via socket-based TCP/IP.

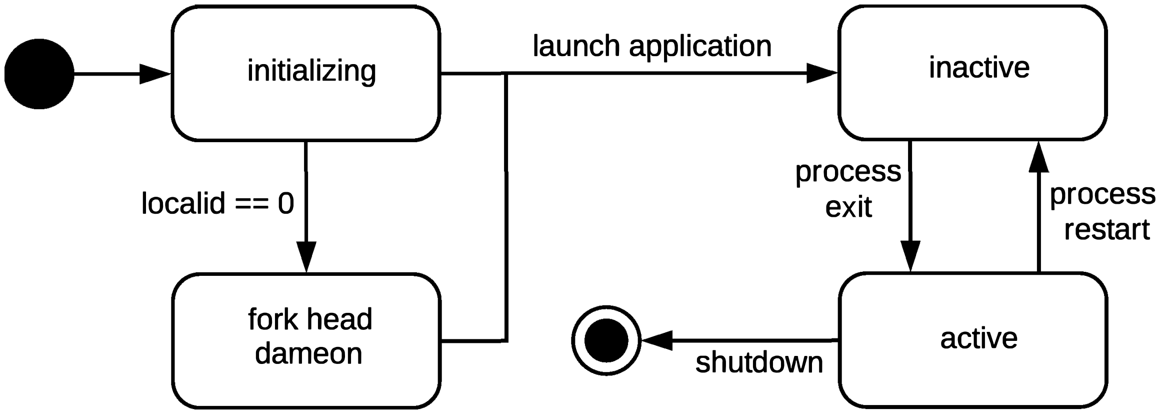


Figure 5.2: State diagram of fault dæmon.

5.5.3 Failure Propagation

A perfect fault detector requires all surviving processes to be notified about a failure. Our MPI runtime propagates the error information to all live processes. Figure 5.5 shows the error propagation mechanism of `MPI Stages`. After launching the MPI processes, fault dæmons enter into "inactive" mode. When an MPI process fails, the corresponding fault dæmon becomes "active" and gets the exit code of the process. If the exit code is non zero, it notifies the head dæmon about the failure, which in turn notifies the controller. The controller dæmon starts the propagation of error. First, it propagates the error information to all nodes (head dæmons). In next step, the head dæmon propagates the failure to the live processes of that node. To promptly notify all live processes about the failure, we need a asynchronous failure propagation mechanism which does not rely on polling. We use signal handler for prompt failure notification. To receive the failure notification, all processes register an error handler after MPI initialization. This error handler lets processes to register a signal handler (SIGUSR2). As soon as the head dæmon gets notified by the controller, it propagates the failure by sending signal to all live processes.

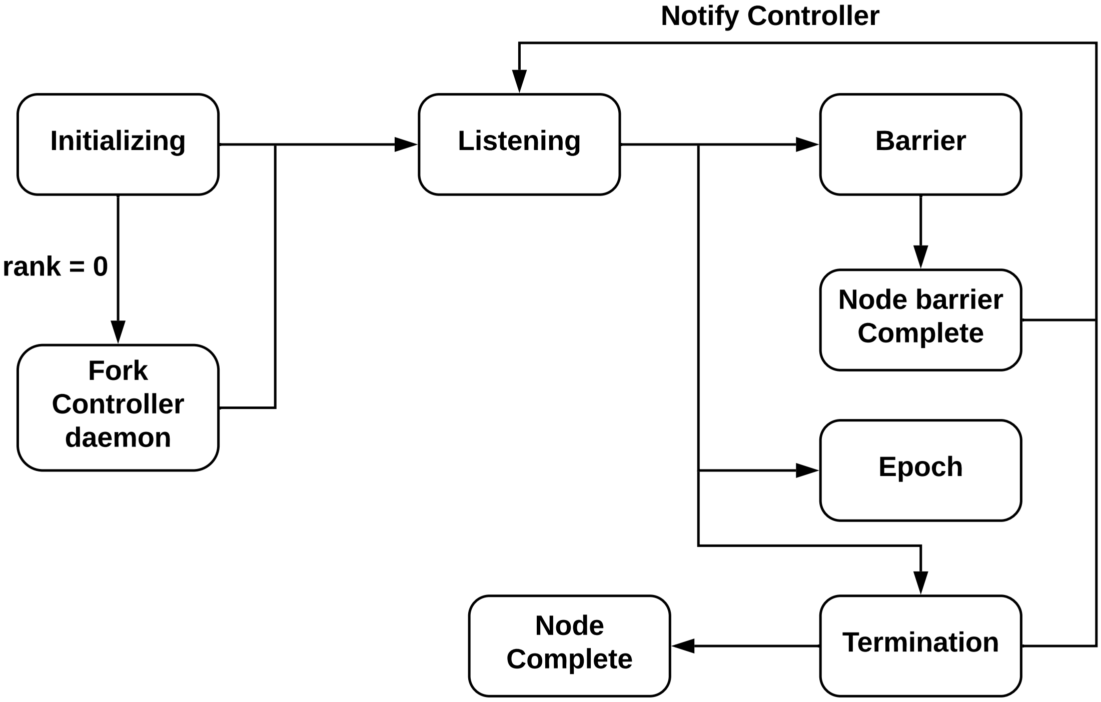


Figure 5.3: State diagram of head dæmon.

5.5.4 Failure Recovery

The two main design aspects of `MPI Stages` are (a) transparent replacement of a failed process by the MPI runtime system and (b) managing and capturing internal MPI state.

The runtime system (fault dæmon) replaces the failed process with a new process and relaunches it (hence, a relaunched process). After the head dæmon gets notified by the controller, it sends a restart request to the active fault dæmons (only the dæmon corresponding to the failed process will be active) to relaunch the failed process.

To distinguish between a first-time start process and a relaunched process, we use a new configuration variable that we denote as an “epoch”. All first-time start processes have epoch value of zero. We increment the value of epoch every time we take a checkpoint. The epoch has meaning both for the application checkpoint and the MPI checkpoint. When `MPI_Init` is called for all first-time start processes with epoch zero, it initializes all MPI state (e.g., communicators, groups).

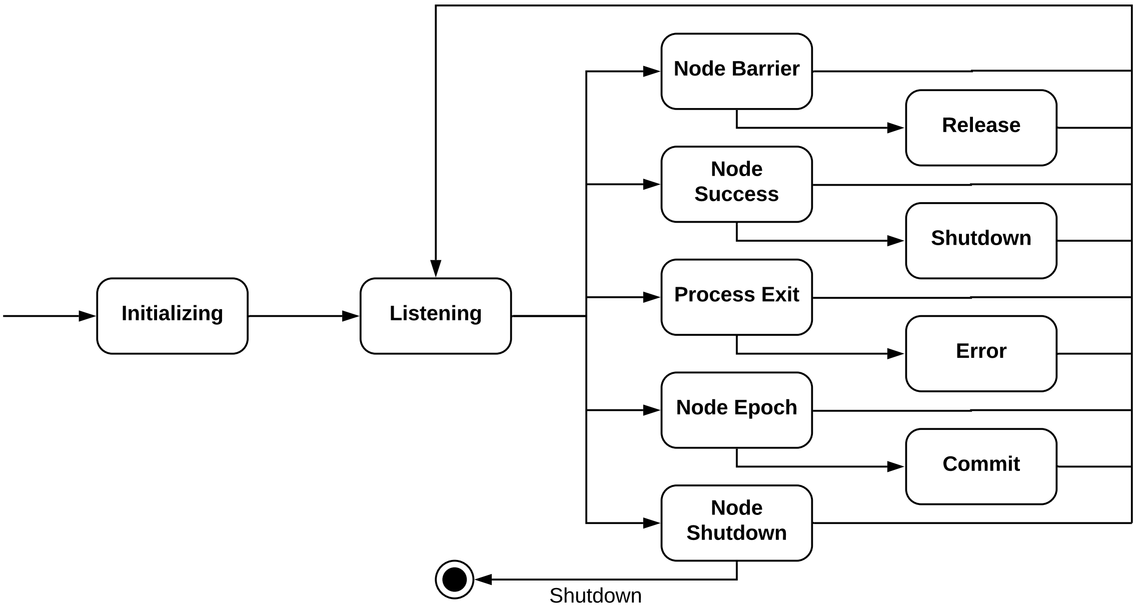


Figure 5.4: State diagram of controller dæmon.

When we relaunch a process after failure, the epoch gets the value of the last synchronous checkpoint. So, the relaunched process invokes `MPI_Init` with an epoch greater than zero. In this case, instead of initializing, `MPI_Init` restores the MPI state from the appropriate MPI checkpoint. Thus, we add overloaded meaning to the `MPI_Init` function based on the value of epoch[3].

For live processes, error handling initiates when runtime triggers the signal handler to notify about a failure. Here, all processes register the pre-defined error handler, `MPI_ERRORS_RETURN`. The error handler cleans up any pending communication (posted requests) and returns an error code `MPIX_TRY_RELOAD` from every MPI function from then on. If there is no pending communication, then the next MPI call will return that error code. The error code tells the application to rollback and restore the last synchronous checkpoint for application state. This allows the application to use a structured approach to in-application exception handling without special language support. Note that relaunched processes eventually become live processes after all recovery actions have been taken.

[3] If this were to be standardized, we might opt for a generalized initialization function that also provides for a non-default error handler as well.

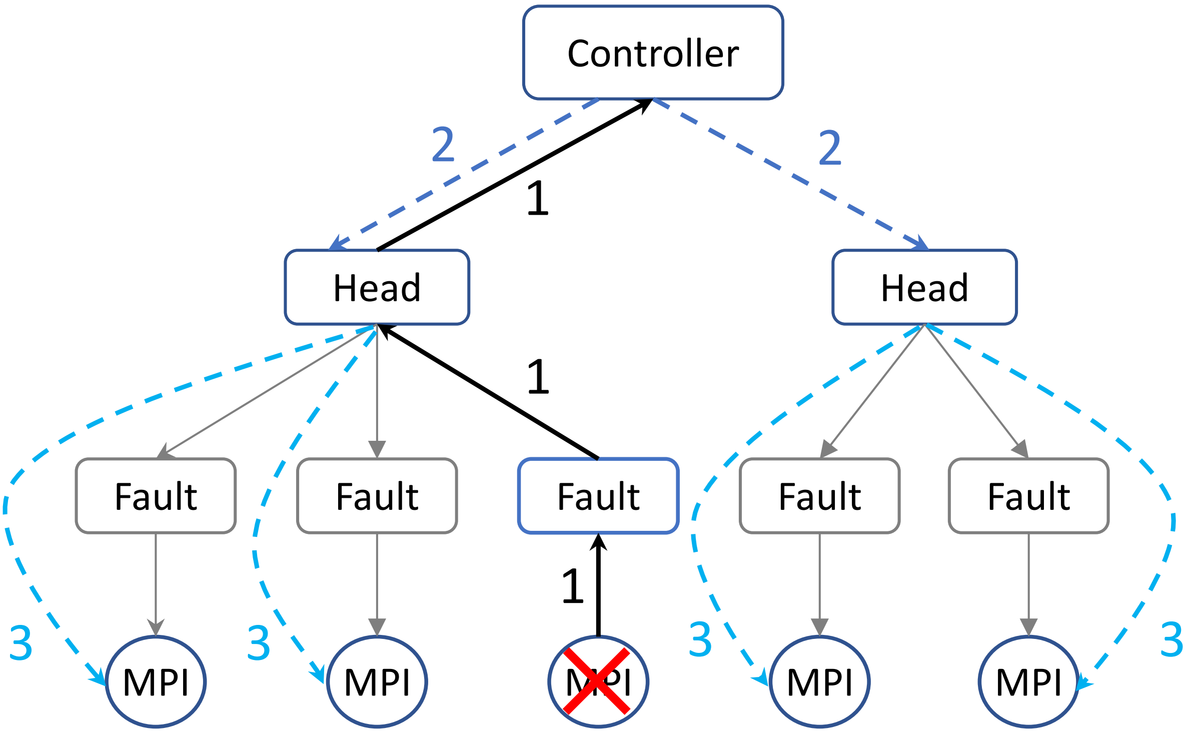


Figure 5.5: Illustration of error detection and propagation in `MPI Stages`.

Figure 5.6 shows the execution flow of the `MPI Stages` recovery among 3 processes. After each stage, the program takes a checkpoint of both MPI and application state and increments the epoch value. Here, process 2 fails during stage3. The live processes (process 0 and process 1) get failure notification, clear pending requests, return error code to the application, and wait for the failed process. On the other hand, a replacement process is launched with the last successful epoch (2) and loads the state of MPI from checkpoint. Then all processes load their application checkpoint and continue execution.

5.5.4.1 Loading Checkpoint

After receiving the error code, the application calls `MPIX_Checkpoint_read()`. All live processes enter into a quiescent state (implicit barrier) and wait for the relaunched process to synchronize. Once the relaunched process completes loading the MPI checkpoint and joins the barrier, they all load the application checkpoint and continue execution. Any messages in transit across processes are lost and will need to be re-sent. The application will have consistency on exactly what has and has not been sent prior to the recovery. Before loading the

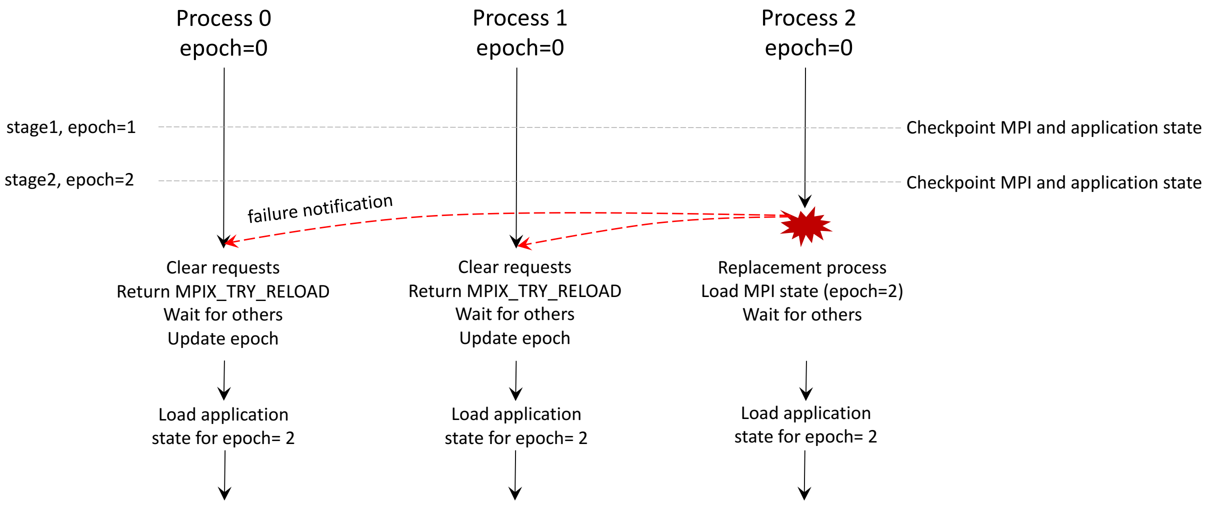


Figure 5.6: Illustration of execution flow of an MPI program using `MPI Stages` recovery.

application checkpoint, the application calls `MPIX_Get_fault_epoch()` which returns the last synchronous checkpoint number to load the application data.

5.5.4.2 Addressing In-Flight Data

A further contribution of our approach is addressing potential in-flight data after a recovery. The epoch value is also used as part of the implementation of transports that support our approach to resilient MPI. Each epoch represents a unique communication phase for the application. Messages from older epochs will be dropped if received later (after recovery). By propagating the epoch[4] in to the message protocol, we eliminate the potential for inconsistency from in-flight data. This resolves a significant issue compared to what ULFM and other systems have had to consider. This epoch approach to versioning the transfers could also be adapted with other fault-tolerant approaches that involve a synchronization and consensus step at recovery.

5.5.4.3 Handling Asymmetrical Error Code

During an execution, some processes may detect a failure while some other may consider the execution was successful. Let's imagine a ring communication among 3 processes starting with process 0. Process 0 sends to process 1 and waiting to receive from process 2, which is waiting

[4] In practice, this currently adds 32 bits to messages (besides the context, source rank, and tag) aiding the matching of messages. If such space is limited, we can make assumptions about the likelihood of wrapping in this field due to many old messages from earlier epochs.

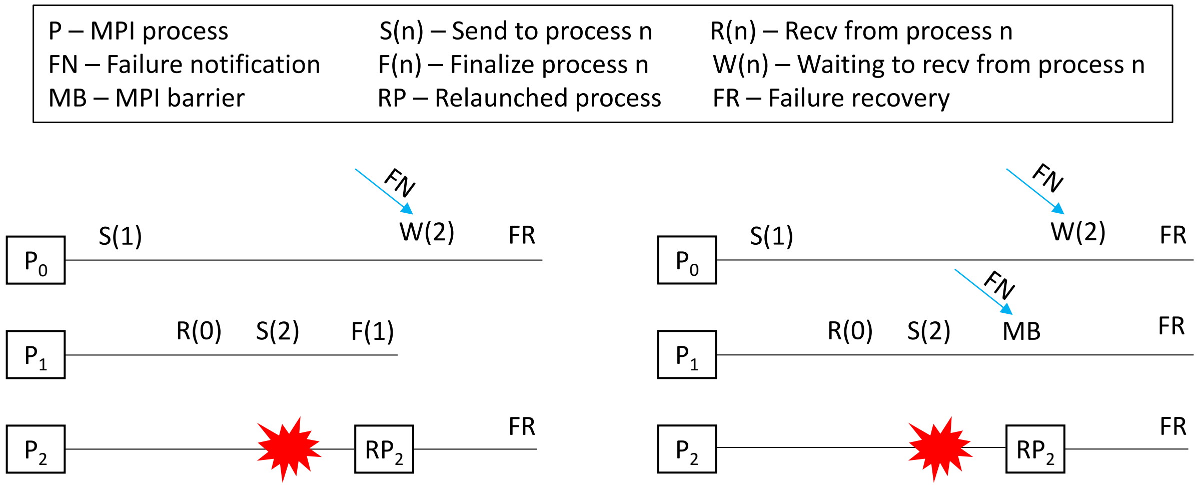


Figure 5.7: Illustration of asymmetrical error handling. The left figure shows that P_0 and P_2 are blocked in recovery phase as P_1 has finalized. The right figure shows the solution to complete the recovery by using a barrier.

to receive from process 1, which is waiting to receive from process 0. After process 1 receives from process 0 and sends to process 2, its involvement in the communication has completed. Process 1 can leave the resilient loop and call `MPI_Finalize`. In the meantime, process 2 has failed, process 2 is relaunched, process 0 is notified; both of them enter into recovery and wait for process 1 to complete the implicit barrier. However, process 1 has already completed the execution. As a result, the processes will be in a deadlock. Figure 5.7 (left) illustrates the above scenario.

To resolve the issue, `MPI Stages` adds an `MPI_Barrier` before returning from `main_loop`, which ensures that no processes will leave the resilient loop until all of the application code is executed. For the above scenario, process 1 waits in the barrier (inside the resilient loop) instead of calling `MPI_Finalize`. When process 2 fails, all of the processes can participate in the recovery. Figure 5.7 (right) shows that all of the processes enter into the failure recovery phase.

A similar scenario may occur in the barrier or some processes may fail during `MPI_Finalize`. Our model cannot recover once some processes are outside of the resilient loop. However, in this case, the application has already completed its computation. `MPI Stages` will abort the execution with proper error messages for the user. Alternatively, we can apply

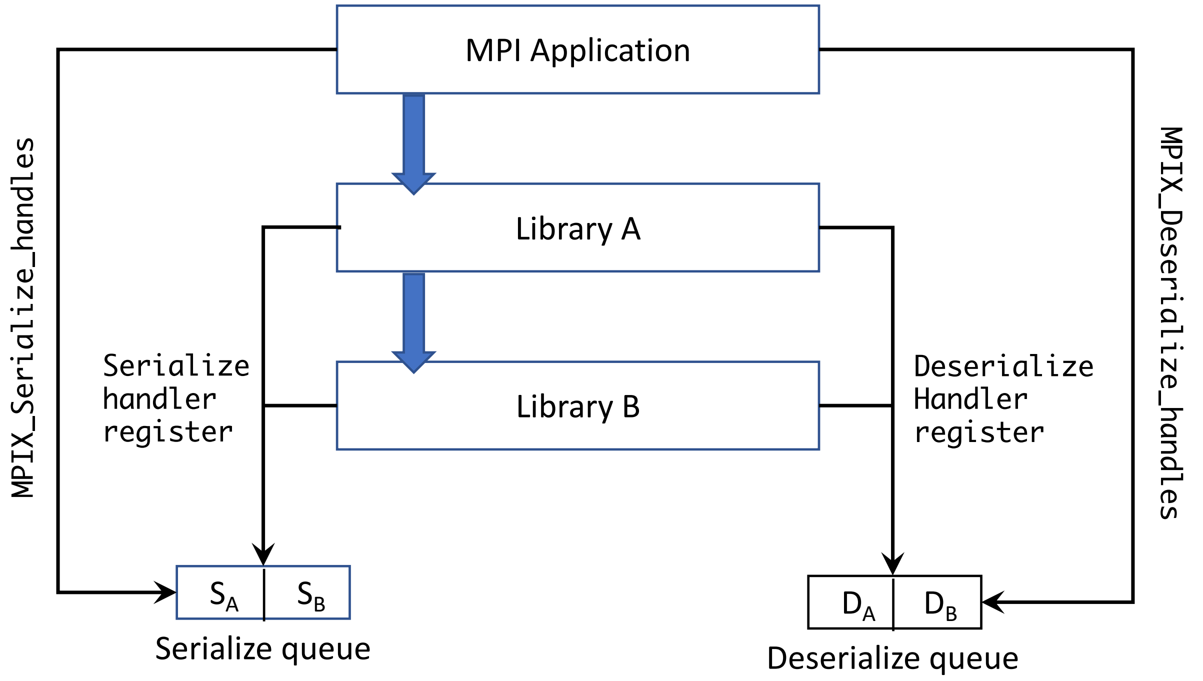


Figure 5.8: Illustration of serialization and deserialization of MPI handles with external libraries.

other recovery methods such as application checkpoint/restart, although the main computation has completed without any failure.

5.5.4.4 Serialization and Deserialization of MPI handles

MPI Stages periodically serializes the state of MPI in a checkpoint using MPIX_Checkpoint_write. This function serializes the state of communicator (e.g., process groups, context id) and state of transport (e.g., network connectivity). The relaunched process restores the state of MPI by deserializing the checkpoint. However, an application may use multiple communicators or groups other than MPI_COMM_WORLD. In this case, to restore the user MPI handles after a failure, we need to serialize the MPI objects referred by these handles along with the MPI state.

In the current form of the model, we use programs that are primarily restricted to using MPI_COMM_WORLD. However, our model supports "re-vivifying" user handles such as other communicators and groups. Because we require that there are no pending requests during the checkpoint—a valid assumption for most bulk synchronous programs that use CPR—we do not need to revivify MPI requests.

An MPI application may use supplemental software libraries. Complex library composition requires special design consideration for any fault-tolerant model. To maintain consistency after failures, we need to revivify any MPI handles these libraries use.

To accomplish this, we introduce callback functions, serialize/deserialize handlers that application and libraries can register. They can register their handlers using `MPIX_Serialize_handler_register` and `MPIX_Deserialize_handler_register` functions.

When an MPI application calls `MPIX_Serialize_handles`, it invokes the serialize handlers in the order they were registered. An `MPIX_Handles` object is passed as an argument to each handler. The handler populates the `MPIX_Handles` object with user MPI handles. After executing the handler, `MPIX_Serialize_handles` serializes all the MPI resources referred by `MPIX_Handles` into MPI checkpoint file. A similar funtion, `MPIX_Deserialize_handles` is used to revivify the user MPI handles after failure. It creates the object `MPIX_Handles` by deserializing the MPI checkpoint file and calls the deserialize handlers.

Figure 5.8 shows the use of software libraries in MPI application. Here, the application calls a library—A and library A calls the second library—B. Both libraries register their serialize handlers (S_A and S_B) and deserialize handlers (D_A and D_B) into separate queues. The MPI library serializes and deserializes the queues when the application calls the corresponding functions.

As described in the MPI Standard, handles can be implemented using different types (e.g., integers or pointers). Our implementation uses integer types for handles. We guarantee application-level semantics for the handles after an MPI checkpoint is reloaded. More formally, if a handle x refers to the MPI resource y (stores in a checkpoint) before a failure, we guarantee that x refers to the same resource y (restores from the checkpoint) after a failure.

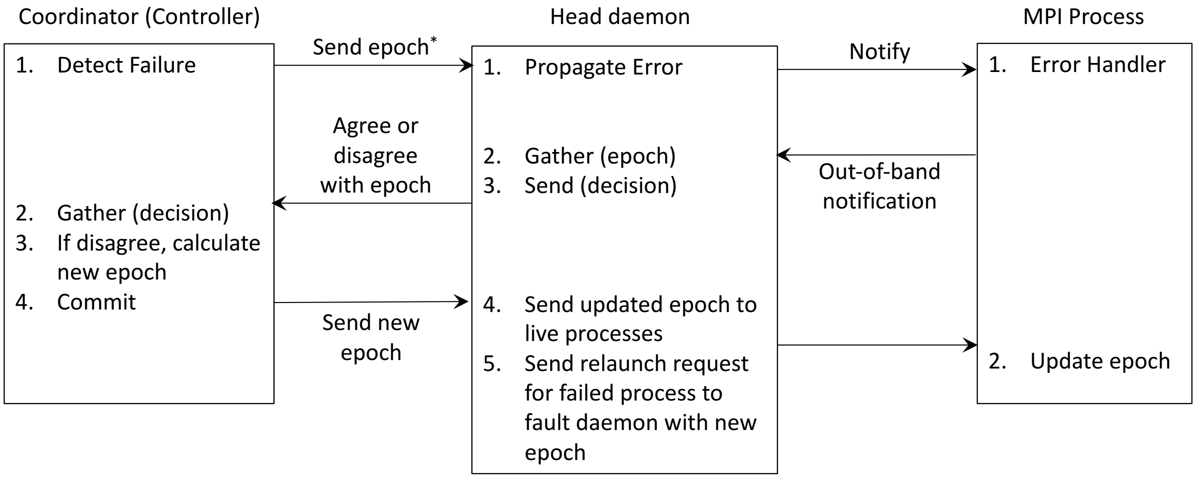


Figure 5.9: Illustration of agree/commit consensus algorithm ($epoch^*$ is the last successful epoch of failed process).

5.5.4.5 Agreement on epoch

In MPI Stages, epoch indicates the last successful checkpoint. After detecting a failure, runtime launches a new process to replace the failed one. This relaunched process loads its MPI state from a checkpoint during MPI initialization. So, the relaunched process needs an epoch that is synchronized among all processes to load the MPI checkpoint.

To accomplish this, we use a consensus algorithm that determines the last synchronous epoch value for all processes. This epoch is also used to load application checkpoint. A consensus algorithm plays an important role in fault tolerant applications [87]. Different algorithms have been introduced for distributed consensus (e.g., two-phase consensus [88]) (see Section 2.4.2). In our model, we use a agree/commit approach of consensus. Figure 5.9 illustrates the agreement algorithm used in MPI Stages.

The runtime controller dæmon acts as the coordinator in the agreement process. It starts the agreement by sending the epoch of the failed process to all head dæmons along with the notification of failure. Head dæmons propagate the error information by notifying the live MPI processes. The live MPI processes send an out-of-band notification to the head dæmons and invoke their error handler. Each head dæmon now gathers the epoch value of the MPI processes it launched and compares it with the epoch of the failed process. If all MPI processes have the same epoch as the failed process, the server sends the decision "agree" to the coordinator. If any MPI process has a different epoch, server sends that epoch to the coordinator along with "disagree". The coordinator gathers all decisions. If the result is unanimous, the coordinator commits the epoch of the failed process to all head dæmons. Otherwise, it calculates the lowest common epoch and commits the new value. Finally, the head dæmon sends the restart command to its active fault dæmon with the updated epoch to replace the failed process. It also sends the epoch to all of the live processes. A call to `MPIX_Get_fault_epoch` will return the synchronized epoch value.

5.6 Sample Fault-Tolerant MPI program

Listing 5.3 shows a sample bulk synchronous MPI program that uses traditional CPR-based recovery method. The program starts with MPI initialization, then initializes or restores application state, and finally starts MPI communications over a simulation loop. After each iteration, application state is saved to a checkpoint. After a fail-stop process failure, the MPI job is restarted, the application initializes MPI, and starts execution by restoring the last application checkpoint.

Listing 5.4 shows the same MPI program as Listing 5.3 but applies `MPI Stages`. In our model the resilient loop in the `main` method runs until the application successfully completes or aborts. First, the resilient loop initializes MPI similar to Listing 5.3. Then it sets an error handler to `MPI_ERRORS_RETURN` and calls the `main_loop` to execute the simulation with an epoch number. The epoch is zero for initial execution and greater than zero if a failure occurs. In the `main_loop`, the application either initializes the state or restores it from a checkpoint. At the end of each iteration, we store the application and MPI state in checkpoints. In the simulation loop, we check the return error code for each MPI call. When a live process receives error code `MPIX_TRY_RELOAD`, it returns to the resilient loop and restores its MPI state to the last checkpoint. Note that this error-communication approach avoids language-specific exception handling and does not require the use of `setjmp/longjmp`. In the meantime, the failed process is re-spawned and also restores its state. Then all processes call the `main_loop`, restore the application state, and continue execution from the iteration that has failed. When the simulation is successfully completed, it terminates the resilient loop.

```
#include<mpi.h>
int main(int argc, char **argv)  {
   /* MPI initialization */
   MPI_Init(&argc, &argv);

   /* Initialize or Restore application state */
   if (checkpoint exists)
      Application_Checkpoint_Read(...);
   else
      /* Initialize application state */

   /* Main simulation loop */
   while (iteration < MAX_ITERATION) {
      /* MPI communications */
      ...
      MPI_Allreduce(...);
      ...
      /* Save application state */
      if (last iteration)
        MPI_Barrier(...);
      else
        Application_Checkpoint_Write(...);
   }
   MPI_Finalize()
   return 0;
}
```

Listing 5.3: Sample program following traditional CPR-based paradigm

In summary, our model sets an error handler, stores MPI state in a checkpoint in tandem with application state, and restores both MPI and application state upon a failure. All processes initialize their MPI state only once. Only the process that has failed and re-spawned need to call `MPI_Init()` again. However, in this case, `MPI_Init` restores the MPI state from a checkpoint instead of through initialization.

5.7 Implementation

Instead of using an existing MPI library—with extremely complex internal state—we implement a prototype of our fault-tolerant model in a new MPI implementation, ExaMPI [14] (see Chapter 4). This allowed us to avoid legacy issues in the code base of existing mature implementations.

```
#include<mpi.h>
int main(int argc, char **argv) {
  int code = MPI_SUCCESS;
  while (not done or abort) { //Resilient loop
    switch(code) {
    case MPI_SUCCESS:
       /* MPI Initialization */
       MPI_Init(&argc, &argv);
       /* Set error handler */
       MPI_Comm_set_errhandler(...);
       break;
    case MPIX_TRY_RELOAD:
       /* Restore MPI state from checkpoint*/
       MPIX_Checkpoint_read();
       break;
    default:
       MPI_Abort(...);
       break;
    }
    /* Get the last synchronous checkpoint# */
    MPIX_Get_fault_epoch(&epoch);
    /* Main simulation loop */
    code = main_loop(argc,argv, epoch, &done);
  }
  MPI_Finalize();
  return 0;
}

int main_loop(int argc,char **argv,int epoch,int *done) {
  /* Initialize or restore application state*/
  if (epoch > 0)
     Application_Checkpoint_Read(...);
  else
     /* Initialize application state */

  /* Main simulation loop */
  while (iteration < MAX_ITERATION) {
     /* MPI communication */
     ...
     code = MPI_Allreduce(...);
     if (code != MPI_SUCCESS)
        return MPIX_TRY_RELOAD;
     ...
     /* Save application state */
     Application_Checkpoint_Write(...);
     /* Save MPI state */
     MPIX_Checkpoint_write();
  }
  /* All other application related code */
  ...
  /*Barrier to ensure no processes call finalize until the simulation
     completes*/
  code = MPI_Barrier(...);
  if (code == MPI_SUCCESS)
     *done = 1; /* Successful completion */
  return code;
}
```

Listing 5.4: Sample fault-tolerant program following with MPI Stages paradigm

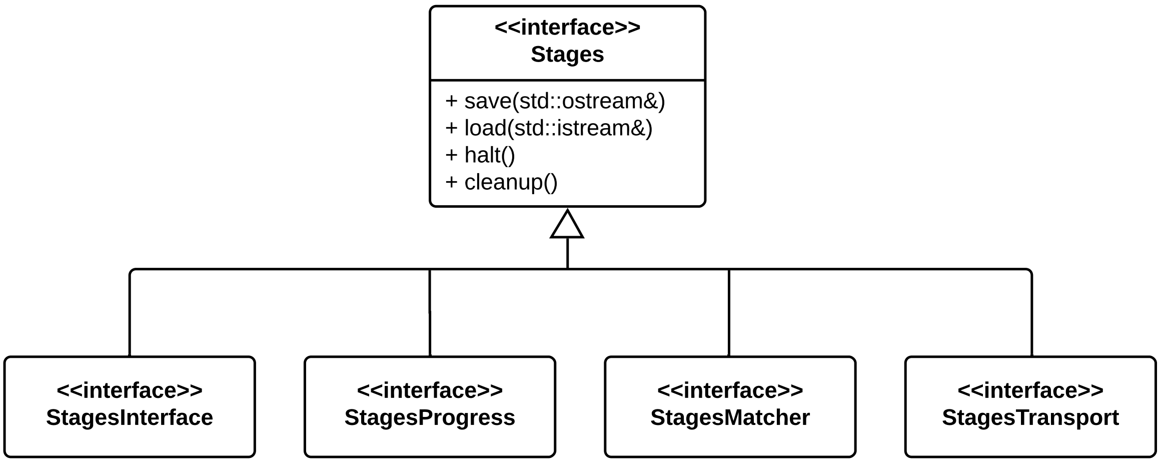


Figure 5.10: Illustration of UML description of Stages interface.

We isolate the new API extensions (StagesBasicInterface) added by MPI Stages (see Listings 5.1 and 5.2) from the standard MPI API as shown in Figure 4.5 (see Section 4.3.4).

We use the command line parameter $--enable\_mpi\_stages$ with mpiexec to enable the `MPI Stages` model.

Our recovery requires to manage and capture the internal state of MPI. To support MPI state checkpointing, we provide a "Stages" interface as shown in Figure 5.10. This interface has four routines— *save*, *load*, *halt*, and *cleanup*. Different modules of the MPI implementation (e.g., Progress, Transport) implement these routines by extending the Stages interface.

The save function writes the state of the MPI into a checkpoint. The load function reads the state of MPI from a particular checkpoint. The `MPIX_Checkpoint_write` and `MPIX_Checkpoint_read` functions call the corresponding save/load implementations of each module to save or load their corresponding MPI state to/from the checkpoint file. We write the checkpoints as a binary file per MPI process. To reduce the overhead of checkpointing, we can adopt different techniques used in the Scalable Checkpoint/Restart (SCR) library [48]. We can cache only the latest checkpoint in the local storage of a node and discard older ones. We can also apply the redundancy schemes used in SCR such as periodically copying a cache checkpoint to the parallel file system or storing a copy of it to another node. We can utilize local RAM disk or solid-state drives (SSDs) for storage based on availability.

To release any pending requests (waiting to receive message), we use the cleanup routine. Finally, the halt is used to set the error code of any posted requests to `MPIX_TRY_RELOAD`.

`MPI Stages` could be implemented in any existing MPI implementation. However, it requires significant engineering effort to capture and manage the internal state of MPI middleware; advance design for this may be crucial.

5.8 Conclusions

We presented a new approach for fault tolerance in MPI, `MPI Stages`. It detects process failure, propagates the error to all live processes, and notifies to the user application. To recover from process failure, `MPI Stages` transparently replaces a failed process by loading its MPI state along with its application state from a saved checkpoint. This allows surviving (live) processes to stay within the main computation loop—close lexically to where failures occurred—while only relaunched processes restart from the beginning of the program. `MPI Stages` introduces a set of API extensions to the MPI Standard. We implemented our model in a prototype MPI implementation (ExaMPI) and presented the requirements to implement this fault tolerance model in a production MPI implementation. The experimental design and evaluation of `MPI Stages` is discussed in Chapter 6.

Chapter 6

Experimental Design and Evaluation of MPI Stages

6.1 Introduction

In this chapter, we show the experimental design and evaluation of `MPI Stages`. We implement a prototype of `MPI Stages` as a proof-of-concept in ExaMPI library (see Chapter 5). We demonstrate its functionality and performance through mini applications and microbenchmarks. We evaluate the recovery time of the test applications using `MPI Stages` and compare its performance with other recovery models. In the following sections, we discuss the test environment and various use cases of our fault-tolerant model.

6.2 Test Environment

In this section, we describe each aspect of the test environment in turn.

6.2.1 Applications

To evaluate the performance and functionality of `MPI Stages`, we test mini applications as well as microbenchmarks. We use LULESH (Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics) [89, 90] and CoMD (Co-design Molecular Dynamics) [91] proxy applications as a representative target bulk synchronous application. We also implement two microbenchmarks— Ring communication and matrix multiplication. We apply `MPI Stages` to LULESH, CoMD, Ring communication, and matrix multiplication. We use the Slurm job scheduling system to run the test applications. Section 6.3 discusses the details of each test application.

6.2.2 Recovery Models

Our model primarily targets applications that use bulk synchronous programming model. The state-of-the-practice recovery model for these applications is checkpoint/restart. The global-restart model, Reinit, is a more suitable recovery model for BSP applications [73]. Also, the closest competitor of `MPI Stages` is the Reinit model. We present the performance of our model by comparing it with checkpoint/restart and the global-restart model, Reinit.

6.2.2.1 Checkpoint/restart

We implement a prototype of checkpoint/restart (CPR) recovery model in ExaMPI library. For each test application, we implement functions to write and read its state to/from a checkpoint file. In traditional CPR, in case of process failure, the runtime kills the entire job and restarts the MPI application by requeuing the job again for allocation. For the purpose of this dissertation, CPR model kills all live MPI processes in case of any process failure and restarts the MPI application again without requeuing the job. In this case, we use "salloc" to obtain a Slurm job allocation in the cluster. We use the same allocation to restart our applications.

6.2.2.2 Reinit

We implement a prototype of the global-restart model, Reinit, in our MPI implementation. In the prototype, we only implement the `MPI_Reinit` function which is the key component of the Reinit interface. We modified our test applications to use Reinit model for recovery. Reinit model uses a checkpoint of the application state during recovery. Here, we reuse the application checkpoint functions developed for checkpoint/restart.

The resilient version of these applications invoke `MPI_Reinit` right after `MPI_Init`. They pass a restart handler to `MPI_Reinit` which is the start point of the program. `MPI_Reinit` sets up the resilient environment and invokes the restart handler. During the reinitialization of live processes, it uses the internal code of `MPI_Finalize` to cleanup MPI resources and `MPI_Init` to initialize MPI again.

The runtime of Reinit spawns a new process after detecting a process failure. However, no agreement is needed before relaunching the new process like `MPI Stages` (see Section 5.5.4.5).

In Reinit, instead of cleaning up and returning an error code to the application, all live processes jump to the restart point. To jump to the restart point from any point in the execution, we use setjmp/longjmp semantics. We run our test applications by submitting a Slurm batch job (“sbatch”) in the cluster.

6.2.3 System

To evaluate our implementation, we use the Quartz cluster with Intel (Xeon E5-2695) 36-core 2.1 GHz nodes, 128 GB of memory per node, Omni-Path interconnect and the Catalyst cluster with Intel (Xeon E5-2695) 24-core 2.4 GHz nodes, 128 GB of memory per node, InfiniBand QDR interconnect at the Lawrence Livermore National Laboratory (LLNL). The Quartz system has 2,634 user-available nodes with 36 cores per node. The Catalyst system consists of 324 nodes with 24 cores per node.

6.2.4 File System

All three recovery models, `MPI Stages`, Reinit, and checkpoint/restart require to save and load application states to/from a checkpoint file. Additionally, `MPI Stages` requires to write and read the states of MPI. To evaluate the performance of checkpoint file storage (e.g., local storage versus parallel file system storage), we use multiple different file systems to store the checkpoint files, and analyze the time required to read or write a checkpoint file in those systems. We use three different file systems— (a) the temporary file system of a node (/tmp), (b) the Linux parallel file system, Lustre (/p/lscratch#), with a capacity of 5 PB and a bandwidth of 90 GB/sec, and (c) the NFS mounted home directory (slower than temporary and parallel file systems) to save our checkpoint files. For `MPI Stages`, each process creates two checkpoint files (application and MPI checkpoint).

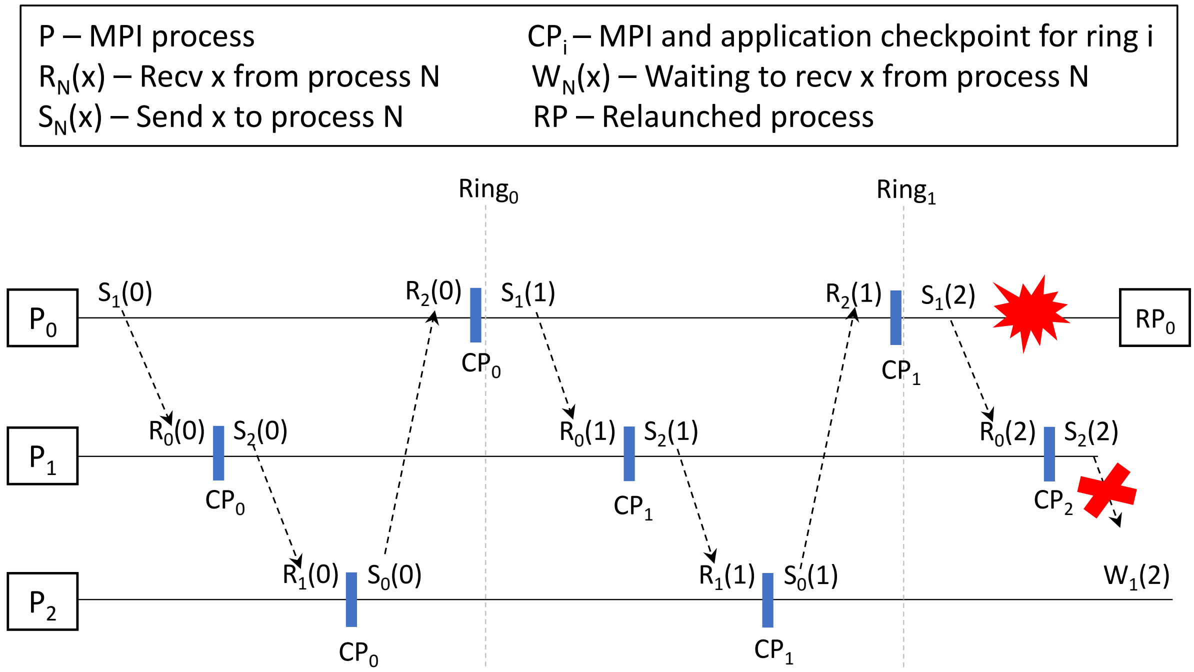


Figure 6.1: Illustration of Ring communication with 3 processes. Here, the process P_0 fails after completion of second ring.

6.3 Use Cases

Here, we discuss various use cases of `MPI Stages`. We evaluate our fault-tolerance model on both standalone MPI applications and library composition.

6.3.1 Ring Communication

The first use case is a microbenchmark, ring communication. In this example, N MPI processes communicate in a circular fashion. Process 0 starts broadcasting the data and gets the same data back from process $N - 1$ (completing a ring); process 0 then modifies the received data and starts the next ring. Figure 6.1 shows Ring communication among three processes.

The Ring starts by sending 0 from P_0 to P_1. After receiving data from P_0, P_1 takes a checkpoint and forwards the same data to P_2. Each process takes a checkpoint after receiving the data when there is no pending requests on that process. After each ring completion, P_0 increments the value by 1. Here, after the completion of the second Ring ($Ring_1$), P_0 checkpoints its application and MPI state, sends next data (2) to P_1, and then fails. When P_1 tries to send the data to P_2 it realizes that a process has failed; whereas P_2 gets notified about the

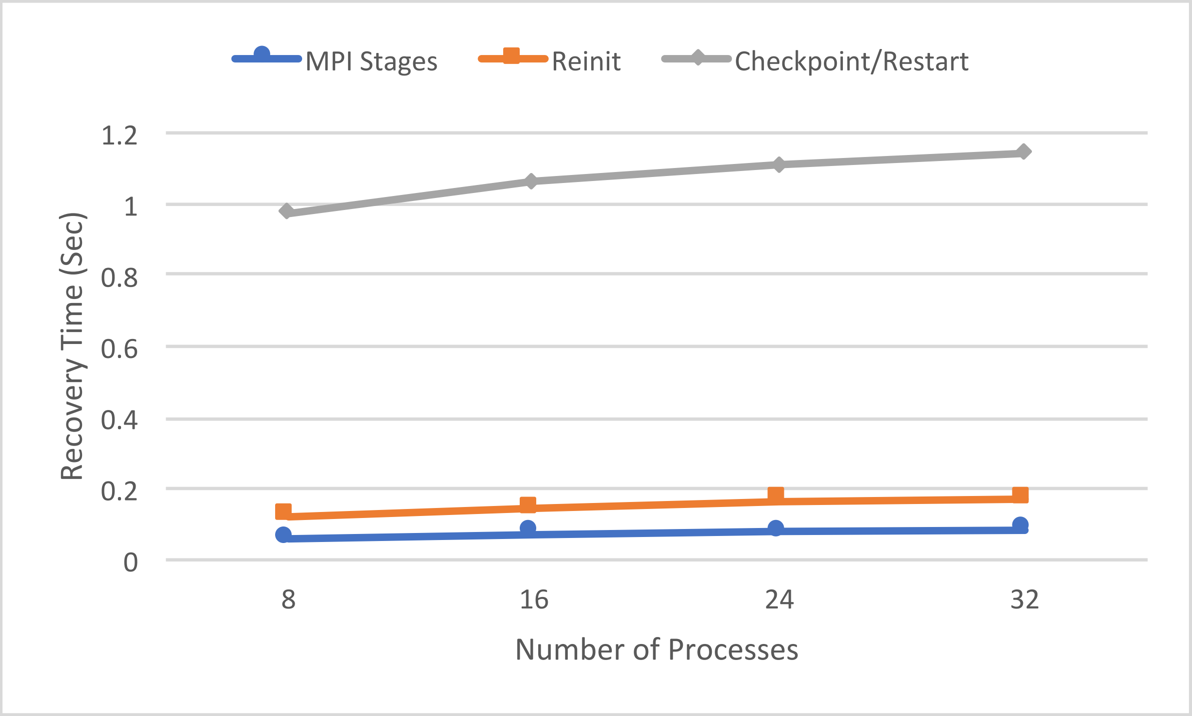


Figure 6.2: Recovery time of `MPI Stages`, Reinit, and CPR for a single process failure in Ring microbenchmark. We observe that `MPI Stages` recovers faster compared to Reinit and CPR.

failure and clears all of its pending requests. Both P_1 and P_2 rollback by returning error code `MPIX_TRY_RELOAD`; P_0 has been re-spawned. The execution resumes from checkpoint CP_1. Although P_1 had a more recent checkpoint CP_2, the last synchronous checkpoint among all processes is CP_1. CP_1 indicates that all processes have successfully completed $Ring_1$. So, the Ring starts again by sending 2 from P_0 to P_1.

We tested the Ring example with different numbers of MPI processes. We compared the recovery time required for the scenario shown in Figure 6.1 using our method, `MPI Stages`, reinit, and traditional checkpoint/restart (CPR). Here, we saved the checkpoints in home directory (as checkpoint size is small). Figure 6.2 presents the recovery time including checkpoint read in seconds for `MPI Stages` and CPR. It shows that the recovery time of `MPI Stages` is at least 11× faster than CPR. However, the recovery time of Reinit is very close to our method as the application initialization time was comparatively small.

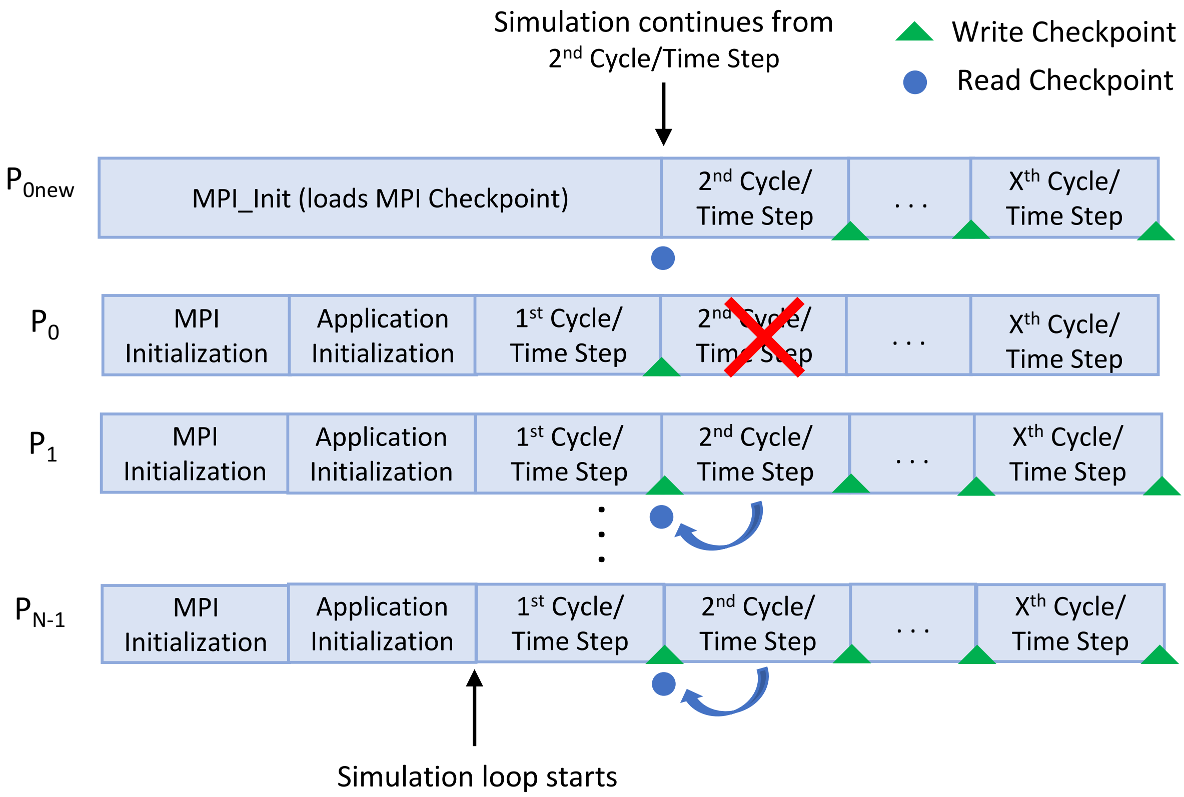


Figure 6.3: Illustration of sample simulation of BSP applications. Here, process P_0 fails during second cycle/time step of simulation loop.

6.3.2 LULESH

LULESH [89, 90, 92] is a simplified proxy application, used to solve a simple Sedov blast problem. However, it represents the program pattern of a real bulk synchronous scientific applications. We used our prototype MPI library to test LULESH (version 2.0.3) in failure-free mode. Then, we applied MPI Stages on it to recover from process failures. We modified the main method of LULESH by adding a "resilient loop" and by moving the main simulation loop into a separate method "main_loop" as mentioned in Section 5.6.

Figure 6.3 shows a sample simulation of a bulk synchronous application using MPI-Stages. Here, N MPI processes (configured as a logical perfect cube of processes as required by LULESH) are participating in a simulation over X iterations. LULESH only uses MPI_COMM_WORLD for communication. In MPI initialization, each process initializes its MPI states. In the application initialization phase, each process sets up the mesh and decomposes,

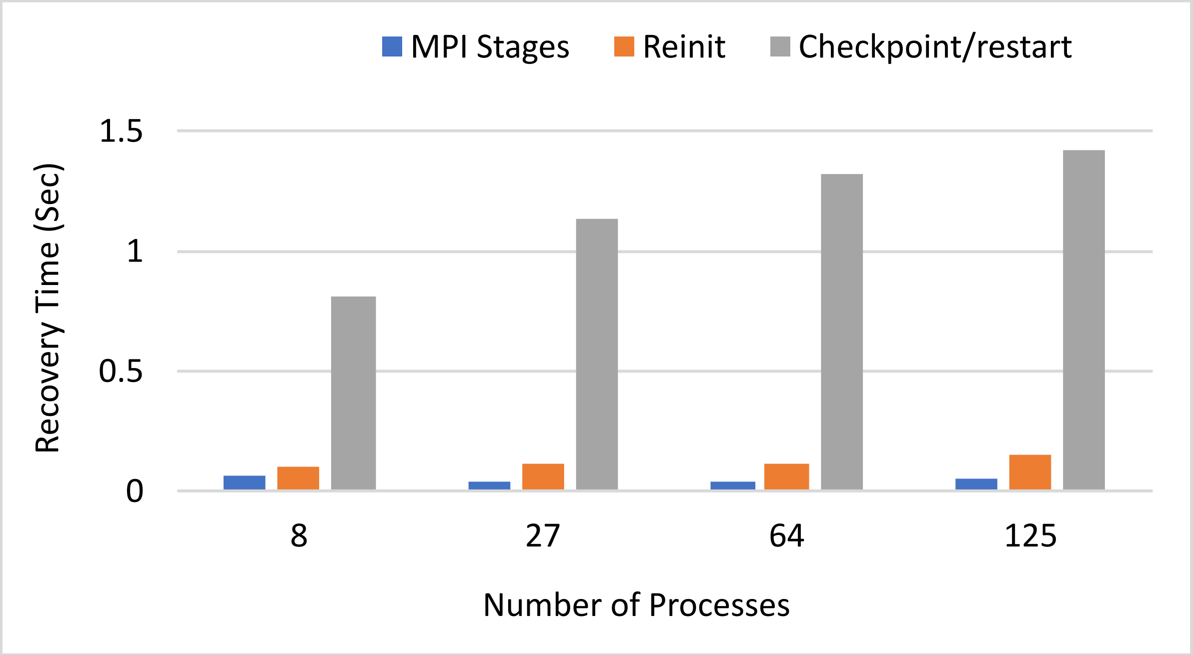


Figure 6.4: Recovery time of MPI Stages, Reinit, and CPR for a single process failure for LULESH (strong scaling with 38^3). We observe that MPI Stages reduces the recovery time compared to both CPR and Reinit.

builds and initializes the main data structure (Domain), and initiates the domain boundary communication. Then the main simulation loop starts.

After the $1^{st} cycle$, all processes save their application and MPI state in a checkpoint and then start the next cycle. In the middle of the $2^{nd} cycle$, P_0 fails. The runtime system detects the failure, spawns a new process P_{0new} to replace P_0. It also triggers the error handler of all the live processes (P_1 to P_{N-1}). The live processes discard all pending requests and rollback to the last checkpoint. In the mean-time, P_{0new} completes MPI_Init by loading the last synchronous MPI state checkpoint. Finally, all processes load their application state from the last checkpoint and continues the simulation from the $2^{nd} cycle$.

We applied MPI Stages, Reinit, and CPR to recover from the process failure (see Figure 6.3). The recovery time using these three approaches is shown in Figure 6.4. Here, for all three approaches, we isolate the recovery time and do not include the time to read the application checkpoint. The reduction in recovery time is significant for MPI Stages as compared to CPR. MPI Stages reduces the recovery time by at least 13×. Also, CPR exhibits a gradual increase in recovery time as the number of processes increase. MPI Stages also reduces

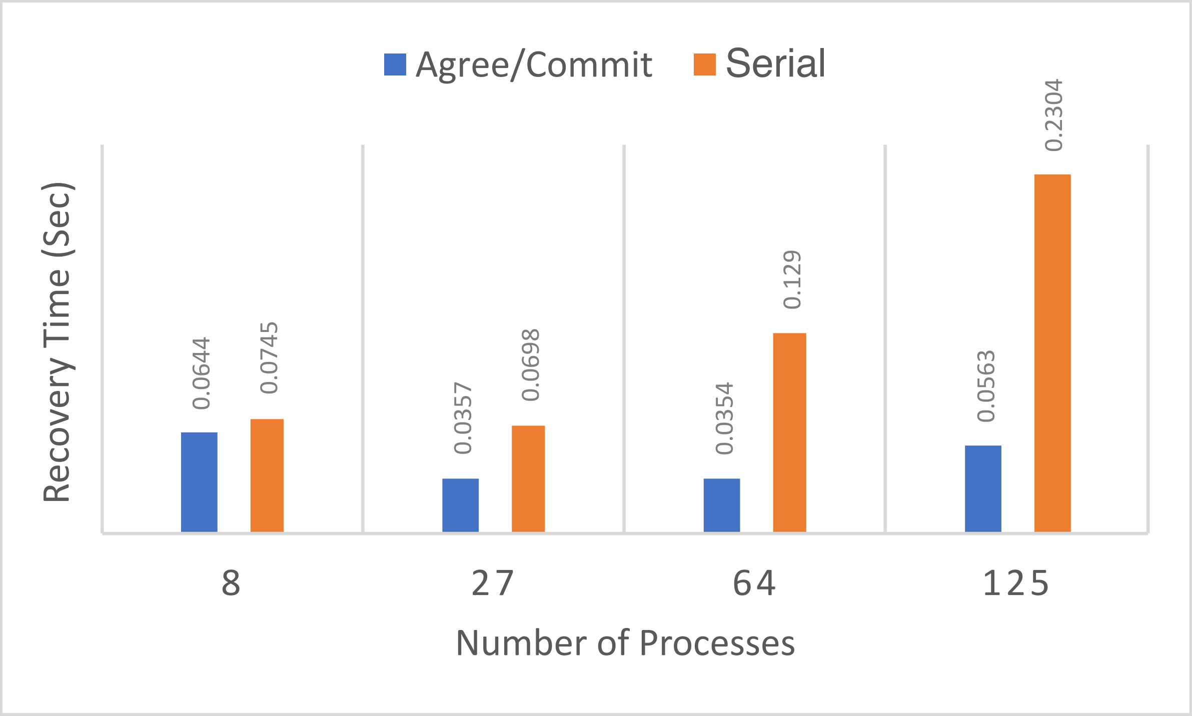


Figure 6.5: Reduction in recovery time using agree/commit consensus.

the recovery time compared to Reinit. As both of the models use global-restart model, the reduction is comparatively smaller. The reduction mainly depends on the initialization phase of the application and the MPI. Since, the recovery time in Reinit depends on both application and MPI initialization time. Figure 6.4 shows that the reduction is almost 2×. Also, we find a very slow increase in recovery time of Reinit as the number of processes increase.

We used an agree/commit consensus approach to determine the synchronous epoch value (see Section 5.5.4.5). The serial algorithm loops through the epoch configuration file of all processes and determines the lowest common value. The new agree/commit consensus algorithm substantially reduces the recovery time in `MPI Stages`. Figure 6.5 compares the recovery time of LULESH using these two approaches.

We ran LULESH with a problem size of 38^3 per domain. Here, we demonstrate the result for strong scaling. For this size, the application checkpoint size is 15MB and the MPI checkpoint size ranges from 544B to 1.3KB. The MPI checkpoint size is significantly smaller than the application checkpoint. The time required to read/write an MPI checkpoint is negligible compared to the application checkpoint.

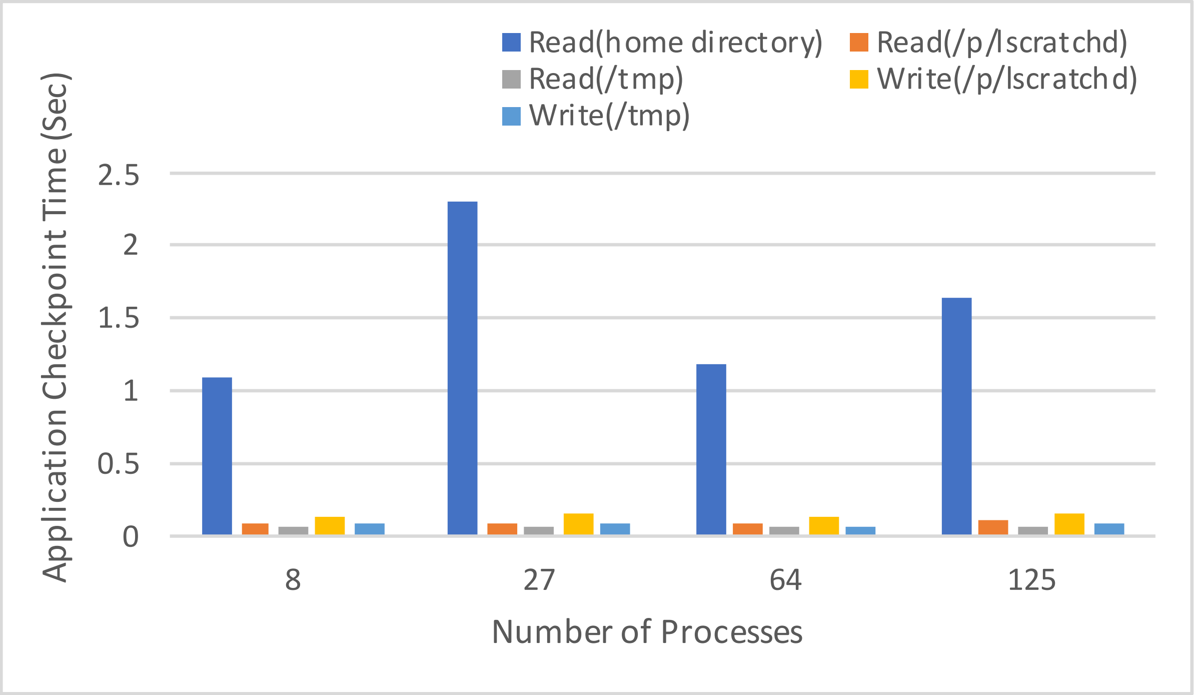


Figure 6.6: Application checkpoint time of LULESH using different file systems.

Figure 6.6 represents the application checkpoint time for different file systems. As expected, the parallel and temporary file systems significantly reduce the checkpoint time compared to the home directory.

Figure 6.7 compares the recovery and checkpoint time for our method. The recovery time is almost always lower than the checkpoint time. In our prototype, we save the checkpoint after each iteration. However, we can trivially reduce this overhead by taking a checkpoint after a certain number of iterations. There are different models to determine the optimal checkpoint interval. It can be derived using a fixed failure rate [93] or using any distribution of time between failures [94]. We can utilize these models to determine the optimal frequency of our checkpoint.

We isolated the time taken by individual functions of MPI Stages interface in LULESH and presented it in Figure 6.8. Here, MPI_Init (First-time) presents the average time to initialize MPI for all first-time start (epoch 0) processes. The MPI_Init for a relaunched process is the time needed to read the MPI state from a checkpoint. Figure 6.8 shows that the initialization of the MPI state takes more time than loading it from a checkpoint. The

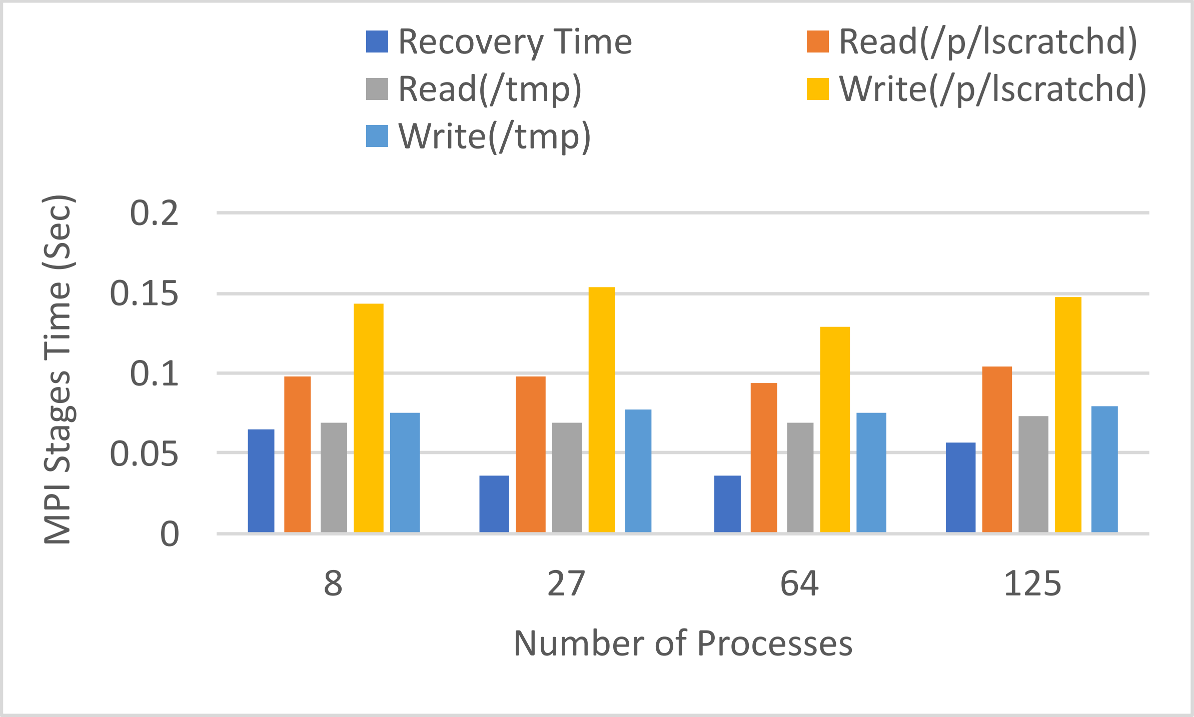


Figure 6.7: Comparison of times for `MPI Stages`.

`MPIX_Checkpoint_read` time indicates the average time spent by live processes to synchronize with the relaunched process (waiting time of live processes). Finally, we show the average time to write a MPI checkpoint. We exclude `MPIX_Get_fault_epoch` as it only performs local access, so the time is negligible.

6.3.3 CoMD

CoMD is a proxy application created by Exascale Co-design Center for Materials in Extreme Environments (ExMatEx) [91]. It is a reference implementation of classical molecular dynamics algorithms and workloads. It provides implementations for calculating simple Lennard-Jones (LJ) and Embedded Atom Method (EAM) potentials. It implements a simple geometric domain decomposition to divide the total problem space into domains, which are owned by MPI ranks. Each domain is a single-program multiple data (SPMD) partition of a larger problem. CoMD uses a single communicator (`MPI_COMM_WORLD`) for the entire job.

The main program of CoMD consists of three blocks—prolog, main loop, and epilog, which is the typical pattern of bulk synchronous applications. The job of prolog is to validate

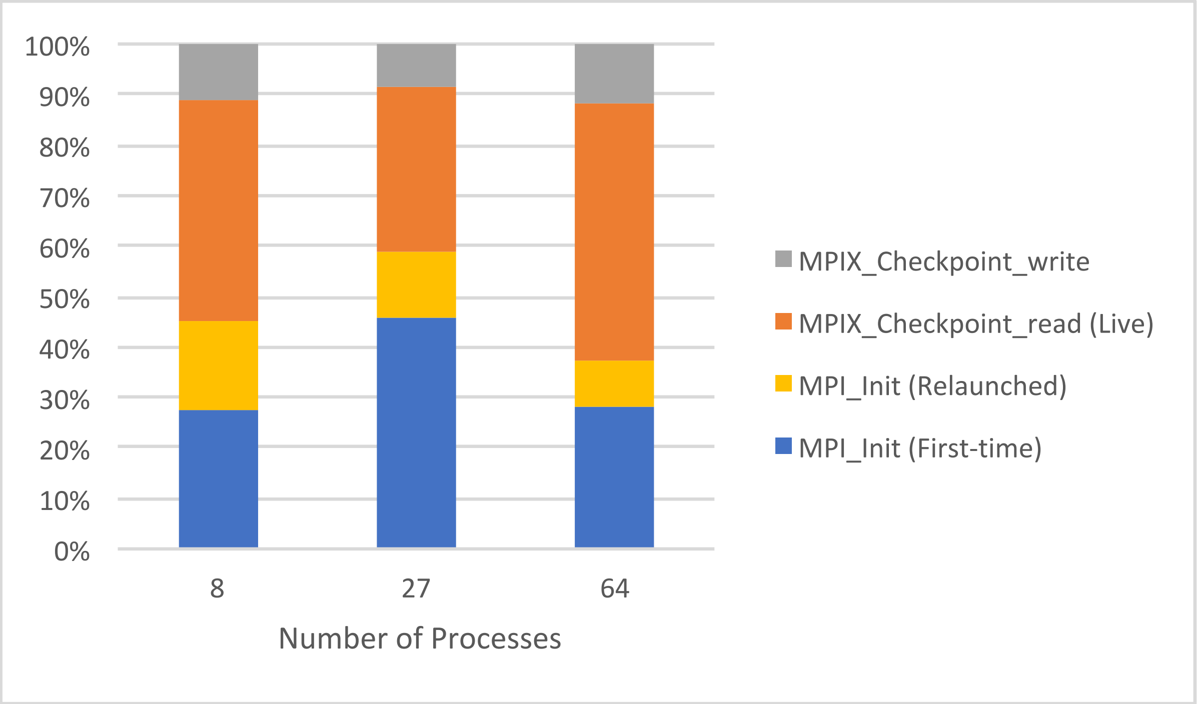


Figure 6.8: Time taken by individual `MPI Stages` interface functions (LULESH strong scaling with 38^3).

the input and initialize the simulation. Then main loop updates the particle positions in a time step simulation by computing forces and communicating atoms between ranks. Finally, the epilog handles validation and clean up of memory.

We modified the main program of CoMD to incorporate `MPI Stages` as shown in Section 5.6. CoMD follows the same simulation pattern shown in Figure 6.3. However, the N MPI processes are configured as a multiple of the ranks in x, y, z direction. In our simulation, all processes save their MPI and application state in a checkpoint after 10 time steps (one iteration).

We applied MPI Stages, Reinit, and CPR to recover form the single process failure (see Figure 6.3). The recovery time using these models is shown in Figure 6.9. The recovery time excludes the time to read an application checkpoint. `MPI Stages` reduces the recovery time by at least 8× compared to CPR.

We executed CoMD for a short run of 100 time steps. In the run, we used the many-body Embedded-Atom Model (EAM) potential as force model. The performance of `MPI Stages` and Reinit is very close for processes 1 to 8. For processes 1 to 8, we used a very small

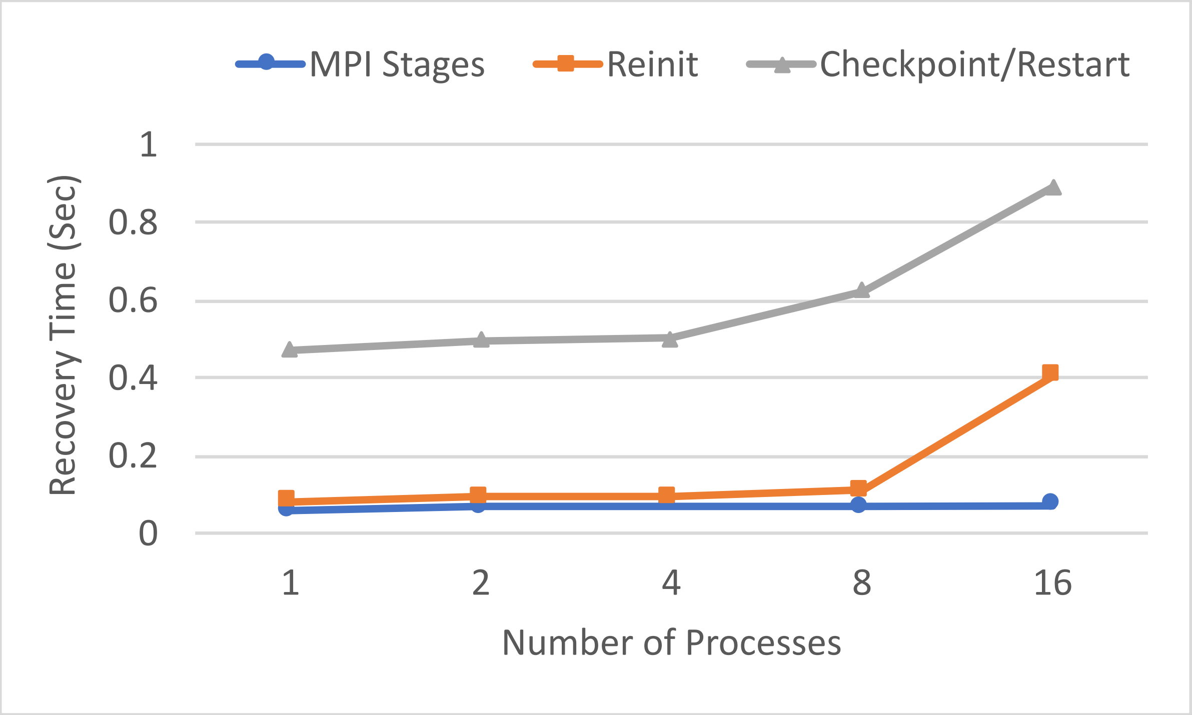


Figure 6.9: Recovery time of MPI Stages, Reinit, and CPR for a single process failure for CoMD with a problem size of 4000 atoms for processes 1 to 8 (strong scaling) and 64000 atoms for 16 processes.

problem size (only $4,000$ atoms). So, the application initialization time was comparatively small. For 16 processes, we used a problem size of $64,000$ ($4,000$ atoms per task). Here, as we increase the problem size, the application initialization time of CoMD increases. So, the recovery time of Reinit increases as well. Since, `MPI Stages` recovery avoids application initialization so it reduces the recovery time by 5×.

6.3.4 Matrix Multiplication Library

We used a micro-benchmark, matrix multiplication (MM), as a use case to demonstrate the use of supplemental software libraries in an MPI application. In this example, we developed a library that multiplies two matrices in parallel using MPI and returns the result to the application. Our MPI application invokes the multiplication function of the library over N iterations. At the end of each iteration, it updates the input matrices with the multiplication result and stores its state in a checkpoint. Then, it starts the next iteration with the updated input matrices.

```
/********** MM Library API **********/
void MM_library_init(MPI_Comm comm);
void MM_library_finalize();
void MM_library_multiply(Matrix matA, Matrix matB, Matrix *result);
void MM_library_callback_register();

/********** Implementation **********/
void MM_library_callback_register() {
  MPIX_Serialize_handler_register(s_handler);
  MPIX_Deserialize_handler_register(d_handler);
}

void s_handler(MPIX_Handles *handles) {
  /* Allocate memory for handles */
  . . .
  MPI_Comm *comms = handles->communicators;
  *(comms++) = mm_comm;
  handles->comm_size++;
  . . .
}
void d_handler(MPIX_handles handles) {
  . . .
  mm_comm = handles->communicators[handles->comm_size - 1];
  handles->comm_size--;
  . . .
}
```

Listing 6.1: API of matrix multiplication library

The library creates its own communicator (mm_comm) by duplicating the communicator passed from the application. The application does not have access to the internal MPI handles (newly created communicator) of the library. However, `MPI Stages` requires the application to revivify this handle during recovery from a failure. We used the serialization/deserialization functionality (see Section 5.5.4.4) to revivify the user handle of MPI object. Listing 5.2 presents the API extensions of MPI added by `MPI Stages` for serialization/deserialization.

Listing 6.1 shows the interface of the matrix multiplication (MM) library. It uses *MM_library_callback_register* method to register the serialize/deserialize handler (*s_handler* and *d_handler*). The matrix multiplication library is not stateful, so we do not need to save any library states other than the MPI handles.

Listing 6.2 shows the pseudocode of the application that uses the matrix multiplication (MM) library. Here, we only show the main_loop.

```
int main_loop(int argc, char **argv, int epoch, int *done) {
    MM_library_init(MPI_COMM_WORLD);
    MM_library_callback_register();
    if (recovery)
        /* Deserialize all MPI handles (both application and libraries) */
        MPIX_Deserailize_handles();
        Application_Checkpoint_Read(...);

    else
        // Initialize application state

     while (iteration < MAX_ITERATION) {
          MM_library_multiply(...);
          ...
          Application_Checkpoint_Write(...);
          MPIX_Checkpoint_write();
          /* Serialize all MPI handles (both application and libraries) */
          MPIX_Serialize_handles();
     }
     ...
}
```

Listing 6.2: Sample Fault-Tolerant Program with MPI handles serialization/deserialization

Application initializes MPI, sets the error handler to MPI_ERRORS_RETURN and invokes *main_loop*. In the main loop, it initializes the matrix multiplication library and lets the library register its serialization/deserialization handlers. The application can also add its own handlers.

At the end of each or N iteration of the simulation, it saves application and MPI state in separate checkpoints. Then, it calls the MPIX_Serialize_handles function that invokes the previously registered serialize handler (*s_handler*) of the matrix multiplication library and passes MPIX_Handles as parameter. The MM library adds its MPI handles (in this case communicator handle) to MPIX_Handles. Finally, MPIX_Serialize_handles serializes the MPI object referred by the handle into the MPI checkpoint.

During iteration X, an MPI process fails while multiplying the matrices in the library. The runtime spawns a new process to replace the failed one and the live processes call their error handler. The error handler cleans up the pending requests and all subsequent MPI calls return MPIX_TRY_RELOAD error code. Library writer could check the error code and return immediately to the application. However, we do not require the external libraries to check this error code. In MPI Stages, applications check the error code of each MPI call. So, the first MPI call after *MM_library_multiply* will check the error code and return to the "resilient loop" (see Section 5.6). The relaunched process deserializes its handles during recovery by calling `MPIX_Deserialize_handles`. This function deserializes the MPI object from MPI checkpoint and creates an `MPIX_Handles` object. Then, it calls the deserialize handler of matrix multiplication library (*d_handler*) which revivifies the communicator handle of the library. Finally, all processes load their application checkpoint and continue execution.

6.4 Conclusions

`MPI Stages` significantly reduces the recovery time of bulk synchronous applications that use checkpoint/restart. We implemented a prototype of MPI Stages in ExaMPI library and demonstrated its functionality and performance on mini applications, LULESH and CoMD and other microbenchmarks. Our results show that MPI Stages reduces the recovery time for both LULESH and CoMD in comparison to the global-restart model, Reinit and traditional checkpoint/restart. Here, we presented the first prototype of `MPI Stages`. Further improvements in the prototype will increase its performance and applicability.

Chapter 7

Future Work

In this dissertation, we developed a tool to tolerate failure in MPI applications that primarily use bulk synchronous programming model. There are several aspects for future direction of this research. One direction could be to improve the current prototype tool to support open source implementations of MPI. Another direction is extending our model to incorporate more programming models, research problems, etc.

One of the fundamental challenge of `MPI Stages` is to define the minimal set of MPI states to checkpoint. In our prototype, we define an MPI state as the current information pertaining to MPI processes, groups, and communicators. We use these as our minimal state to checkpoint ExaMPI, the MPI library used to prototype our model. However, production quality open source implementations of MPI (e.g., OpenMPI, MPICH) have many more complex internal states (e.g., derived data types, network state). Further research is required to identify the MPI states to checkpoint in open source MPI implementations. Significant engineering effort and advanced design will be required to capture and manage the internal state of MPI middleware.

In this work, we primarily target bulk synchronous programming model. Our global, non-shrinking, rollback recovery model fits perfectly with this programming model. However, applications that follow master-slave programming model might not be a good candidate for our recovery model. In this case, application can easily discard the failed process and continue execution with fewer number of processes. We can provide different MPI constructs for error propagation and recovery to support multiple recovery models.

A large number of MPI applications use a hybrid (MPI + OpenMP) programming model. In `MPI Stages`, the rollback starts after an MPI call returns `MPIX_TRY_RELOAD` error code. Consider an application that spends most of its time in computation inside parallel loop and only communicate with other processes using MPI at the end of computation to aggregate the results. In case of a process failure, other processes might spend a lot of time in computation as they are only checking the MPI return code after the computation. This will waste a lot of computation time as those computations will need to be repeated after rollback. A possible solution could be to introduce routine to test for faults. Even in this case, application cannot break from the parallel region as it's illegal in OpenMP. Further research is required on how to handle hybrid application (e.g., cleanup OpenMP before rollback).

One-sided (RMA) operations have been introduced in MPI-2 Standard. However, only a handful of applications actually use this feature. Current version ExaMPI does not have support for MPI RMA operations. `MPI Stages` only addresses two-sided MPI operations per epoch. Fault tolerance in one-sided operations is challenging because of the difficulty of invalidating the RMA windows and handling of in-flight messages. Further research is required to add fault tolerance support to one-sided operations in `MPI Stages`.

The ExaMPI library implements only a subset of MPI-3 functionality. The idea is to extend it by incrementally adding different MPI features as per user requirements. To demonstrate the applicability of `MPI Stages` in more complex production applications, we need to implement different MPI functionality (e.g., full support of MPI communicator). Another important future research aspect of this work is to compose multiple fault-tolerant models into ExaMPI. Apart from single models such as ULFM or checkpoint/restart, we are not aware of successful integration of multiple models. Both the syntax and semantics of such combined models are of interest, but implications for MPI middleware architecture are also of tremendous consequence. ExaMPI library can be used to explore how to manage the complementary, at-times conflicting, and otherwise independent impacts on an MPI implementation arising from multiple models, including how to manage conflict resolutions between multiple models. Also, composition of multiple fault models could help to limit the effect of a failure into a smaller scope.

One of the challenges in adopting new fault tolerance techniques is to modify existing code. `MPI Stages` requires all MPI calls to check for error code which might be impractical for large code bases. One solution could be to use refactoring tool to add error checking.

After detecting a process failure, we do not kill the remaining live processes. This gives us the opportunity for in-memory (diskless) checkpointing. Current prototype of `MPI Stages` uses temporary file system of a node and parallel file system to checkpoint application and MPI states. Instead, we can checkpoint data in memory. In this case, we need to adopt a buddy system to save data into at least two different processors memory. In-memory checkpoint takes advantage of the high speed interconnect and avoids slower I/O operations.

Chapter 8

Conclusions

As machine size increases, the failure rate is also anticipated to increase. Fault tolerance in exascale has become a major concern in HPC community. Most HPC applications use MPI as their programming model. However, the MPI Standard has limited support of fault tolerance. Thus, MPI applications require better fault tolerance support to continue running in exascale machines. Applications use different techniques to recover from a failure. When an MPI program experiences a failure, the most common recovery approach is to restart all processes from a previous checkpoint and to re-queue the entire job (checkpoint/restart). A disadvantage of this method is that, although the failure occurred within the main application loop, *live* processes must start again from the beginning of the program, along with new *replacement* processes—this incurs unnecessary overhead for live processes.

To avoid such overheads and concomitant delays, we introduce the concept of "MPI Stages". MPI Stages saves internal MPI state in a separate checkpoint in conjunction with application state. Upon failure, both MPI and application state are recovered, respectively, from their last synchronous checkpoints and continue without restarting the overall MPI job. Live processes roll back only a few iterations within the main loop instead of rolling back to the beginning of the program, while a replacement of failed process restarts and reintegrates, thereby achieving faster failure recovery. This approach integrates well with large-scale, bulk synchronous applications and checkpoint/restart. We identify requirements for production MPI implementations to support state checkpointing with MPI Stages, which includes capturing and managing internal MPI state and serializing and deserializing user handles to MPI objects.

To implement a prototype of `MPI Stages`, we introduce ExaMPI, a new, experimental implementation of the MPI Standard. ExaMPI solves the problem of full-scale open source MPIs—which are quite large and complex—substantially raises the cost and complexity of proof-of-concept activities. By enabling researchers with ExaMPI, we seek to accelerate innovations and increase the number of new experiments and experimenters, all while expanding MPI's applicability.

Importantly, the results of this work will create middle-out requirements both on networking infrastructure for recoverability from faults, and on the structure of the MPI middleware itself. Through our test implementation, we are able to capture and manage MPI state, something that existing production middleware (such as MPICH and Open MPI) have not had to manage and book-keep carefully. A potential outcome of this work will be specific design requirements for next-generation production MPI middleware that is conscious of state as well as networking infrastructure (device drivers and networks) that enable return to a consistent state after faults and failures.

References

[1] Top 500 Supercomputing site. https://www.top500.org. 2018.

[2] Bianca Schroeder and Garth A Gibson. Understanding failures in petascale computers. In *Journal of Physics: Conference Series*, volume 78, page 012022. IOP Publishing, 2007.

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

[4] Franck Cappello, Al Geist, William Gropp, Sanjay Kale, Bill Kramer, and Marc Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, 2014.

[5] Franck Cappello. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *The International Journal of High Performance Computing Applications*, 23(3):212–226, 2009.

[6] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, page e4851, 2017.

[7] M Emani, I Laguna, K Mohror, N Sultana, and A Skjellum. Checkpointable MPI: A Transparent Fault-Tolerance Approach for MPI. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.

[8] N Sultana, S Farmer, A Skjellum, I Laguna, K Mohror, and M Emani. Designing a reinitializable and Fault Tolerant MPI Library, 2017. Poster presented at EuroMPI/USA 2017, Chicago, IL.

[9] Nawrin Sultana, Anthony Skjellum, Ignacio Laguna, Matthew Shane Farmer, Kathryn Mohror, and Murali Emani. MPI Stages: Checkpointing MPI State for Bulk Synchronous Applications. In *Proceedings of the 25th European MPI Users' Group Meeting*, page 13. ACM, 2018.

[10] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Ignacio Laguna, and Kathryn Mohror. Failure recovery for bulk synchronous applications with MPI stages. *Parallel Computing*, 84:1–14, 2019.

[11] Nawrin Sultana, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. Understanding the Usage of MPI in Exascale Proxy Applications.

[12] Ignacio Laguna, Kathryn Mohror, Nawrin Sultana, Martin Rüfenacht, Ryan Marshall, and Anthony Skjellum. A Large-Scale Study of MPI Usage in Open-Source HPC Applications. In *Proc. of SC 2019*, November 2019. Accepted, in press.

[13] Nawrin Sultana, Martin Rüfenacht, Anthony Skjellum, Purushotham Bangalore, Ignacio Laguna, and Kathryn Mohror. Understanding the Use of MPI in Exascale Proxy Applications. *Concurrency and Computation:Practice and Experience*, 2019. Submitted (Minor revision) September 2019.

[14] Anthony Skjellum, Martin Rüfenacht, Nawrin Sultana, Derek Schafer, Ignacio Laguna, and Kathryn Mohror. ExaMPI: A Modern Design and Implementation to Accelerate Message Passing InterfaceInnovation. *Concurrency and Computation:Practice and Experience*, 2019. Submitted (Minor revision) September 2019.

[15] Message Passing Interface Forum. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. High-Performance Computing Center Stuttgart, University of Stuttgart, 2015.

[16] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[17] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 97–104. Springer, 2004.

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

[19] Cray MPI. https://pubs.cray.com/content/S-2529/17.05/xctm-series-programming-environment-user-guide-1705-s-2529/mpt.

[20] Intel MPI Library. https://software.intel.com/en-us/mpi-library, Aug 2018.

[21] IBM Spectrum MPI. https://tinyurl.com/yy9cwm4p.

[22] Dhabaleswar K Panda, Karen Tomko, Karl Schulz, and Amitava Majumdar. The MVAPICH project: Evolution and sustainability of an open source production quality MPI library for HPC. In *Workshop on Sustainable Software for Science: Practice and Experiences, held in conjunction with Intl Conference on Supercomputing (WSSPE)*, 2013.

[23] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[24] Laxmikant V Kale and Sanjeev Krishnan. CHARM++: a portable concurrent object oriented system based on C++. In *ACM Sigplan Notices*, volume 28, pages 91–108. ACM, 1993.

[25] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[26] Vaidy S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: practice and experience*, 2(4):315–339, 1990.

[27] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23–29, Oct 2000.

[28] Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranarayana. A Survey of Scientific Software Development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 12:1–12:10, New York, NY, USA, 2010. ACM.

[29] Prakash Prabhu, Hanjun Kim, Taewook Oh, Thomas B Jablin, Nick P Johnson, Matthew Zoufaly, Arun Raman, Feng Liu, David Walker, Yun Zhang, et al. A survey of the practice of computational science. In *SC'11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2011.

[30] Steven P VanderWiel, Daphna Nathanson, and David J Lilja. Complexity and performance in parallel programming languages. In *Proceedings Second International Workshop on High-Level Parallel Programming Models and Supportive Environments*, pages 3–12. IEEE, 1997.

[31] David E Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E Grant, Thomas Naughton, Howard P Pritchard, Martin Schulz, and Geoffroy R Vallee. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, page e4851, 2017.

[32] Rolf Rabenseifner. Automatic MPI counter profiling of all users: First results on a CRAY T3E 900-512. In *Proceedings of the message passing interface developer's and user's conference*, volume 1999, pages 77–85, 1999.

[33] Jeffrey S Vetter and Andy Yoo. An empirical performance evaluation of scalable scientific applications. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 16–16. IEEE, 2002.

[34] Sameer S Shende and Allen D Malony. The TAU parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[35] Philip Carns, Robert Latham, Robert Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 characterization of petascale I/O workloads. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[36] Sudheer Chunduri, Scott Parker, Pavan Balaji, Kevin Harms, and Kalyan Kumaran. Characterization of MPI usage on a production supercomputer. In *Characterization of MPI Usage on a Production Supercomputer*, page 0. IEEE, 2018.

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

[38] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *International Workshop on Distributed Algorithms*, pages 126–140. Springer, 1997.

[39] Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 55–70. Springer-Verlag, 2009.

[40] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, (3):219–228, 1983.

[41] Joshua Hursey, Thomas Naughton, Geoffroy Vallee, and Richard L Graham. A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In *European MPI Users' Group Meeting*, pages 255–263. Springer, 2011.

[42] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142. ACM, 1981.

[43] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[44] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[45] Brian Randell. System structure for software fault tolerance. *Ieee transactions on software engineering*, (2):220–232, 1975.

[46] Yi-Min Wang, Pi-Yu Chung, In-Jen Lin, and W. Kent Fuchs. Checkpoint space reclamation for uncoordinated checkpointing in message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):546–554, 1995.

[47] Georg Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of International Conference on Parallel Processing*, pages 526–531. IEEE, 1996.

[48] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis*, pages 1–11. IEEE Computer Society, 2010.

[49] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. FTI: high performance fault tolerance interface for hybrid systems. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[50] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. C3: A system for automating application-level checkpointing of MPI programs. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 357–373. Springer, 2003.

[51] James S Plank, Micah Beck, Gerry Kingsley, and Kai Li. *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994.

[52] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[53] Rohan Garg, Gregory Price, and Gene Cooperman. MANA for MPI: MPI-Agnostic Network-Agnostic Transparent Checkpointing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 49–60. ACM, 2019.

[54] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

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

[56] Lorenzo Alvisi, Bruce Hoppe, and Keith Marzullo. Nonblocking and orphan-free message logging protocols. In *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*, pages 145–154. IEEE, 1993.

[57] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. Redesigning the message logging model for high performance. *Concurrency and Computation: Practice and Experience*, 22(16):2196–2211, 2010.

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

[59] Lorenzo Alvisi. Understanding the message logging paradigm for masking process crashes. Technical report, Cornell University, 1996.

[60] George Bosilca, Aurélien Bouteiller, Elisabeth Brunet, Franck Cappello, Jack Dongarra, Amina Guermouche, Thomas Herault, Yves Robert, Frédéric Vivien, and Dounia Zaidouni. Unified model for assessing checkpointing protocols at extreme-scale. *Concurrency and Computation: Practice and Experience*, 26(17):2772–2791, 2014.

[61] Algirdas Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.

[62] Miguel Castro, Barbara Liskov, et al. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[63] Cijo George and Sathish Vadhiyar. Fault tolerance on large scale systems using adaptive process replication. *IEEE Transactions on Computers*, 64(8):2213–2225, 2014.

[64] Thomas Ropars, Arnaud Lefray, Dohyun Kim, and André Schiper. Efficient process replication for MPI applications: sharing work between replicas. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 645–654. IEEE, 2015.

[65] Graham E Fagg and Jack J Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In *European Parallel Virtual Machine/Message Passing Interface Users Group Meeting*, pages 346–353. Springer, 2000.

[66] Joshua Hursey, Richard L Graham, Greg Bronevetsky, Darius Buntinas, Howard Pritchard, and David G Solt. Run-through stabilization: An MPI proposal for process fault tolerance. In *European MPI Users' Group Meeting*, pages 329–332. Springer, 2011.

[67] Wesley Bland, Aurelien Bouteiller, Thomas Herault, George Bosilca, and Jack Dongarra. Post-failure recovery of MPI communication capability: Design and rationale. *The International Journal of High Performance Computing Applications*, 27(3):244–254, 2013.

[68] Wesley Bland, Aurelien Bouteiller, Thomas Herault, Joshua Hursey, George Bosilca, and Jack J Dongarra. An evaluation of user-level failure mitigation support in MPI. In *European MPI Users' Group Meeting*, pages 193–203. Springer, 2012.

[69] Wesley B Bland. Toward message passing failure management. 2013.

[70] Keita Teranishi and Michael A Heroux. Toward local failure local recovery resilience model using MPI-ULFM. In *Proceedings of the 21st European MPI Users' Group Meeting*, page 51. ACM, 2014.

[71] Amin Hassani, Anthony Skjellum, and Ron Brightwell. Design and evaluation of FA-MPI, a transactional resilience scheme for non-blocking MPI. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 750–755. IEEE, 2014.

[72] Amin Hassani. *Toward a scalable, transactional, fault-tolerant message passing interface for petascale and exascale machines*. PhD thesis, The University of Alabama at Birmingham, 2016.

[73] Ignacio Laguna, David F Richards, Todd Gamblin, Martin Schulz, Bronis R de Supinski, Kathryn Mohror, and Howard Pritchard. Evaluating and extending user-level fault tolerance in MPI applications. *The International Journal of High Performance Computing Applications*, 30(3):305–319, 2016.

[74] Marc Gamell, Daniel S Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. Exploring automatic, online failure recovery for scientific applications at extreme scales. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 895–906. IEEE Press, 2014.

[75] Rajeev Thakur, Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, and Jesper Larsson Träff. MPI at Exascale. *Procceedings of SciDAC*, 2:14–35, 2010.

[76] Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing Lusk, Rajeev Thakur, and Jesper Larsson Träff. MPI on millions of cores. *Parallel Processing Letters*, 21(01):45–60, 2011.

[77] Exascale computing project. https://www.exascaleproject.org. September, 2017.

[78] Exascale Computing Project Proxy Apps Suite. https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/. October, 2017.

[79] Robert D Falgout and Ulrike Meier Yang. hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pages 632–641. Springer, 2002.

[80] HDF5 Support Page. https://portal.hdfgroup.org/display/HDF5.

[81] Rossen Petkov Dimitrov. Overlapping of communication and computation and early binding: Fundamental mechanisms for improving parallel performance on clusters of workstations. 2002.

[82] Rossen Dimitrov and Anthony Skjellum. Software architecture and performance comparison of MPI/Pro and MPICH. In *International Conference on Computational Science*, pages 307–315. Springer, 2003.

[83] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[84] *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015*. 2015.

[85] Ignacio Laguna, David F. Richards, Todd Gamblin, Martin Schulz, and Bronis R. de Supinski. Evaluating User-Level Fault Tolerance for MPI Applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 57:57–57:62, New York, NY, USA, 2014. ACM.

[86] Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, and Manish Parashar. Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 895–906, 2014.

[87] Joshua Hursey, Thomas Naughton, Geoffroy Vallee, and Richard L Graham. A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI. In *European MPI Users' Group Meeting*, pages 255–263. Springer, 2011.

[88] Michael Barborak, Anton Dahbura, and Miroslaw Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys (CSur)*, 25(2):171–220, 1993.

[89] Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 Updates and Changes,number = LLNL-TR-641973. Technical report, August 2013.

[90] Ian Karlin, Abhinav Bhatele, Jeff Keasler, Bradford L. Chamberlain, Jonathan Cohen, Zachary DeVito, Riyaz Haque, Dan Laney, Edward Luke, Felix Wang, David Richards, Martin Schulz, and Charles Still. Exploring Traditional and Emerging Parallel Programming Models using a Proxy Application. In *27th IEEE International Parallel & Distributed Processing Symposium (IEEE IPDPS 2013)*, Boston, USA, May 2013.

[91] Jamaludin Mohd-Yusof, Sriram Swaminarayan, and Timothy C Germann. Co-design for molecular dynamics: An exascale proxy application, 2013.

[92] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[93] John Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *International Conference on Computational Science*, pages 3–12. Springer, 2003.

[94] Yudan Liu, Raja Nassar, Chokchai Leangsuksun, Nichamon Naksinehaboon, Mihaela Paun, and Stephen L Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–9. IEEE, 2008.

Appendix

Appendix A

MPI Functions implemented in ExaMPI

```
int MPI_Abort(MPI_Comm, int);
int MPI_Allreduce(const void *, void *, int, MPI_Datatype, MPI_Op,
    MPI_Comm);
int MPI_Barrier(MPI_Comm);
int MPI_Bcast(void *, int, MPI_Datatype, int, MPI_Comm);
int MPI_Comm_dup(MPI_Comm, MPI_Comm *);
int MPI_Comm_rank(MPI_Comm, int *);
int MPI_Comm_set_errhandler(MPI_Comm, MPI_Errhandler);
int MPI_Comm_size(MPI_Comm, int *);
int MPI_Finalize(void);
int MPI_Get_count(MPI_Status *, MPI_Datatype, int *);
int MPI_Init(int *, char ***);
int MPI_Initialized(int *flag);
int MPI_Irecv(void *, int, MPI_Datatype, int, int, MPI_Comm, MPI_Request
    *);
int MPI_Isend(const void *, int, MPI_Datatype, int, int, MPI_Comm,
    MPI_Request *);
int MPI_Ibsend(const void *, int, MPI_Datatype, int, int, MPI_Comm,
    MPI_Request *);
int MPI_Irsend(const void *, int, MPI_Datatype, int, int, MPI_Comm,
    MPI_Request *);
int MPI_Issend(const void *, int, MPI_Datatype, int, int, MPI_Comm,
    MPI_Request *);
int MPI_Send_init(const void *, int, MPI_Datatype, int, int, MPI_Comm,
    MPI_Request *);
int MPI_Bsend_init(const void *, int, MPI_Datatype, int, int, MPI_Comm,
    MPI_Request *);
int MPI_Rsend_init(const void *, int, MPI_Datatype, int, int, MPI_Comm,
    MPI_Request *);
int MPI_Ssend_init(const void *, int, MPI_Datatype, int, int, MPI_Comm,
    MPI_Request *);
int MPI_Recv_init(void *, int, MPI_Datatype, int, int, MPI_Comm,
    MPI_Request *);
int MPI_Reduce(const void *, void *, int, MPI_Datatype, MPI_Op, int,
    MPI_Comm);
int MPI_Request_free(MPI_Request *request);
```

```c
int eMPI_Recv(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Status * status);
int MPI_Send(const void *, int, MPI_Datatype, int, int, MPI_Comm);
int MPI_Rsend(const void *, int, MPI_Datatype, int, int, MPI_Comm);
int MPI_Ssend(const void *, int, MPI_Datatype, int, int, MPI_Comm);
int MPI_Bsend(const void *, int, MPI_Datatype, int, int, MPI_Comm);
int MPI_Sendrecv(const void *, int, MPI_Datatype, int, int, void *, int, MPI_Datatype, int, int, MPI_Comm, MPI_Status *);
int MPI_Start(MPI_Request *);
int MPI_Wait(MPI_Request *, MPI_Status *);
int MPI_Waitall(int, MPI_Request[], MPI_Status[]);
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
double MPI_Wtime(void);
```