5-2013

Toward Message Passing Failure Management

Wesley B. Bland
wbland@eecs.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Wesley B. Bland entitled "Toward Message Passing Failure Management." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack J. Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

James S. Plank, Gregory D. Peterson, Vasilios Alexiades

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

5-2013

Toward Message Passing Failure Management

Wesley B. Bland
wbland@eecs.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Wesley B. Bland entitled "Toward Message Passing Failure Management." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Jack J. Dongarra, Major Professor

We have read this dissertation and recommend its acceptance:

James S. Plank, Gregory D. Peterson, Vasilios Alexiades

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Toward Message Passing Failure Management

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Wesley B. Bland

May 2013

This dissertation is dedicated to my very supportive wife Julie.

Acknowledgements

I would like to thank the members of the Innovative Computing Laboratory and its support staff for all of the opportunities that they have provided to me, and students like me. I would specifically like to thank my research group, George Bosilca, Thomas Herault, Aurelien Bouteiller, Stephanie Moreaud, Teng Ma, Peng Du, Wes Alvaro, and Blake Haugen for their direction and advice throughout my studies. I would also like to thank Sam Crawford for his many efforts of editing support. To my family, who have provided support, advice, and understanding, I cannot thank you enough and will continue to covet all of those things in my future endeavors.

I see God in the instruments and mechanisms that work reliably.

–Buckminster Fuller

Abstract

As machine sizes have increased and application runtimes have lengthened, research into fault tolerance has evolved alongside. Moving from result checking, to rollback recovery, and to algorithm based fault tolerance, the type of recovery being performed has changed, but the programming model in which it executes has remained virtually static since the publication of the original Message Passing Interface (MPI) Standard in 1992. Since that time, applications have used a message passing paradigm to communicate between processes, but they could not perform process recovery within an MPI implementation due to limitations of the MPI Standard. This dissertation describes a new protocol using the exiting MPI Standard called Checkpoint-on-Failure to perform limited fault tolerance within the current framework of MPI, and proposes a new platform titled User Level Failure Mitigation (ULFM) to build more complete and complex fault tolerance solutions with a true fault tolerant MPI implementation. We will demonstrate the overhead involved in using these fault tolerant solutions and give examples of applications and libraries which construct other fault tolerance mechanisms based on the constructs provided in ULFM.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As High Performance Computing (HPC) passes petascale and moves on to exascale, new challenges have emerged which necessitate a change in the way large scale operations are designed. As of the November 2012 Top500 list [7], machines at the top of the list have now surpassed the million-core mark. Based on the foreseeable limits of the infrastructure costs, an Exaflop-capable machine is expected to be built from gigahertz processing cores, with thousands of cores per computing node, thus requiring millions more computing cores to reach the needed level of performance. At this scale, reliability becomes a major concern. 2007 reliability, availability, and serviceability data analyzed by Schroeder and Gibson [55] show an average number of failures per year between 100 and 1000, depending on the system. Later projections found in Cappello's paper [21] predict a future mean time to failure (MTTF) of approximately one hour. With failures of that frequency, capability applications will not be able to complete without considering a model for handling hardware failures. Table 1.1 shows the machine size and runtime of the Linpack benchmark on some of the the top 10 machines in the November 2012 Top500 list. This data shows that while cores counts increase, runtimes approach and sometimes surpass the 24 hour mark for capacity applications. Machines with a shorter time to completion demonstrate an interesting new trend toward including accelerators in HPC machines; however,

Table 1.1: Machine Size and Linpack runtime on top machines of Top500

Rank	Machine Name	Number of Processors	Runtime (hours)	Tflop/s
2	Sequoia	1,572,864	23.13	16,324.8
3	K Computer	705,024	29.47	10,510.0
5	JUQUEEN	393,216	11.85	4,141.2
6	SuperMUC	147,456	9.00	2,897.0
7	Stampede	204,900	1.56	2,660.2
8	Tianhe-1A	186,368	3.37	2,566.0

they also distort the number of cores counted for the purposes of the list. In reality, the number of processors is higher when accelerator cores are included and therefore the mean time to failure is decreased again. The implications of applications running longer than one day with mean time to failures plunging to an hour are not positive for future HPC productivity.

Capability workloads are not the only applications which motivate the drive for fault tolerance. Current, long running applications already have reached running times of multiple days or weeks on smaller scale machines. Even in this scenario, the likelihood of encountering a failure is non-negligible. For the sake of continued **scalability**, both in terms of numbers of processors and execution time, applications need to be able to continue executions despite hardware failures.

Beyond traditional high performance computing environments, other new areas of distributed computation have also emerged which produce similar needs. Volatile resources such as cloud and grid computing environments have been considered unsuitable for more traditional distributed computing models because of their constantly changing set of resources. If the underlying programming model could support the kind of drop in, drop out behavior that volatile environments need, they could become a lower cost set of tools available for developers. Previous tools (such as HTCondor [58]) have provided an environment for these types of applications, but many codes are already implemented in MPI and the cost of porting the applications to another environment is viewed as too high to warrant the move.

Another area which could benefit from a resiliency model is energy efficient computing. By allowing applications to continue execution beyond failures, expensive recalculations become unnecessary, saving both energy and computation hours. For both of these computing models, a programming model needs to be **dynamic** to support new types of computing.

As will be discussed in Chapter 2, the need for resilience at scale is not a new discovery and has been proven through many previous studies. However, an important factor required for wide adoption, which is often ignored, is **usability**. Many previous efforts to introduce fault tolerance methods into high performance computing have gone un-utilized because they were either too difficult to use or required large changes to existing codes. For any tool to be employed, it must not only fulfill the need of the community, but also be compatible with the other existing tools. Developers need to be able to add fault tolerance into their existing codes with as little disruption as possible.

1.1 New Fault Tolerant Approaches

Previously, the problems discussed above were solved by employing strategies such as transparent rollback recovery or explicit checkpoint/restart (both synchronous and asynchronous). These solutions were sufficient because the bottlenecks that are now becoming hindrances to performance were not yet limiting factors. Now, as algorithms strive to reinvent themselves by creating self-healing techniques, they need a communication library which can provide performance and portability to support them.

In this work, we provide two new fault tolerance models from which application and library developers can choose to solve these problems. We use the de-facto programming environment for parallel applications, the Message Passing Interface (MPI), to provide a familiar, portable, and high performing programming paradigm on which users can base their work. The new models are called Checkpoint-on-Failure (CoF) and User Level Failure Mitigation (ULFM).

CoF CoF is an MPI-3 standard compliant method of providing a form of fault tolerance which employs traditional checkpointing schemes but does so using an optimal number of checkpoints, therefore vastly improving the amount of overhead over traditional periodic checkpointing methods.

ULFM ULFM is a new chapter proposed for the MPI Standard which introduces a new set of tools to create fault tolerant applications and libraries by allowing the applications themselves to design their recovery methods and control them from the user level, rather than an automatic form of fault tolerance managed by the operating system or communication library itself.

These two models are designed to serve different types of applications. CoF is for specific classes of applications which need all processes to be available at all times. Many of these applications might currently be using checkpoint/restart style fault tolerance to resolve failures but incur a large overhead from periodically writing checkpoints to disk, waiting through a batch queue after a failure impacts the application, and restarting the application from the checkpoint.

CoF resolves each of these issues by removing unnecessary checkpoints, maintaining a functional runtime layer to prevent jobs from being reinserted into the batch queue, and allowing checkpoints to stay in local scratch space, possibly even in memory, to improve both checkpoint and restart times.

On the other hand, ULFM is designed to be a solution for a more broad set of applications. Rather than providing a specific solution to insert fault tolerance into applications, ULFM is a platform on which many fault tolerance solutions can be built. By implementing a fault tolerant library on top of ULFM, applications can utilize a portable fault tolerance solution which functions with any MPI implementation that follows the specification.

1.2 Dissertation Statement

The goal of the dissertation is to demonstrate novel methods of fault tolerance supporting Algorithm Based Fault Tolerance (ABFT) for large scale systems using the message passing paradigm with extensions to facilitate concurrent approaches to cope with failures. By allowing applications to continue execution after a hardware failure, algorithms can support larger scale and longer running executions which were previously found to be unattainable.

1.3 Outline

This dissertation will describe the new tools developed to handle failures at the application level while evaluating their performance impact against a failure agnostic MPI implementation. Chapter 2 will provide some background for current and classical research including a survey of existing parallel computing and fault tolerance tools. Chapter 3 will outline the design goals of the fault tolerance techniques described in this dissertation. Chapter 4 will introduce and evaluate the CoF method of failure management. Chapter 5 will describe the ULFM proposal and its implementation along with an evaluation of the overhead introduced and an analysis of an application which uses the new ULFM constructs. Chapter 6 will describe how some existing fault tolerance techniques could be adapted to use ULFM. Finally, Chapter 7 will summarize the research and discuss ongoing and future work related to this dissertation.

Chapter 2

Background & Related Work

Fault tolerance and message passing communication libraries are not new ideas in high performance computing. Fault tolerance began as a solution for unreliable hardware, specifically when building Network of Workstation (NoW) clusters where the systems were not high quality machines. As machines evolved and became more reliable, the challenge shifted from the reliability of a specific piece of hardware to the reliability of the system as a whole. The scale of current HPC machines necessitates the study of new methods of fault tolerance to continue execution on functional machines, while excluding failed machines from applications. In this chapter, we will explore previous efforts in these areas to provide adequate context for the subsequently presented work.

2.1 Terminology

Whenever a discussion of fault tolerance takes place, the terminology being used must first be well established, as many words have evolved to have overlapping definitions. For the purposes of this dissertation, these words will be defined as follows:

Fault – A fault, or error, occurs when some defect, whether hardware or software, is detected. Examples of faults include memory errors due to radiation, data corruption on a hard disk, or a programming error which causes a program to crash.

Failure – A failure is caused by an error when the system cannot mitigate the results of a fault and no longer functions correctly. When a machine ceases to function and the system cannot resolve the issue by automatically repairing the error, it causes a failure.

Fail-stop Fault – A fail-stop fault is a fault which impacts one or more processes and will never be repaired without intervention from the application. An example would be component failure or a total system failure, such as loss of power. In this scenario, the failure cannot be resolved without outside intervention, such as hardware replacement or power restoration.

Transient Fault – A transient fault occurs periodically, but is resolved without intervention from the application. While the fault is causing the system to conclude that a failure has occurred, it is indistinguishable from a fail-stop fault. The most common form of transient fault is a slow network connection, which causes a process to conclude that another process has failed, later to discover that the process is still alive, but could not communicate within the expected window of time.

Byzantine Fault – A Byzantine fault causes unpredictable, often undetectable failures by causing the program to behave incorrectly, but not necessarily to stop functioning. An example of a Byzantine fault is to have malicious software purposely attempt to cause a system to behave incorrectly, or to have a memory corruption error change the system state from one valid state to another valid, but incorrect, state.

2.1.1 Failure Model

For the purposes of this dissertation, we will not consider transient or Byzantine faults, only fail-stop faults. Transient faults should be treated as fail-stop faults and be prevented from further participating in an application. Byzantine faults are much more difficult to mitigate and doing so is outside the scope of this work, though research has shown that managing generic Byzantine faults is NP-hard [45].

2.2 Message Passing Interface

The Message Passing Interface (MPI) [59] is a standardized set of routines written by its governing body, the MPI Forum, used to simplify communication between processes in a parallel application by adopting a message passing abstraction. Processes construct messages using the routines found in the MPI Standard. These messages are sent through an MPI implementation which provides such communication structures as point-to-point messaging, collective communication operations, reduction operations, and more recently, process management, one-sided communication, and file I/O.

Periodically, the standard is updated to adopt new technologies as they become more mature and make corrections to previous versions. Version 1.0 was published on May 5, 1994. In 1995 and 1997, versions 1.1 and 1.2, respectively, were published with corrected errata from the previous versions. Also in 1997, version 2.0 of the standard added many new features, in particular, process creation and management, one-sided communication, new collective communication operations, external interfaces, and parallel I/O. Minor edits and errata were corrected in versions 1.3, 2.1, and 2.2, released in 2008 and 2009. In September of 2012, the most recent version, 3.0 was published to add nonblocking collectives, new one-sided communication operations, and new Fortran bindings to the standard. The MPI Forum is currently convening to discuss MPI 3.1 and MPI 4.0 for future release.

Many implementations of the MPI Standard have been written to fulfill a wide range of purposes. Some, such as Cray MPI, are tightly coupled with a particular type of hardware to run most efficiently and have proprietary code which is not open to the public. Others, such as Open MPI [8, 38], MPICH [4, 18], MVAPICH [5, 41] and older implementations such as LAM/MPI [19, 56], are designed to run on a variety of hardware and are open source, allowing modification by any user who wishes to add a feature or fix a bug. Some implementations, such as FT-MPI [34] and Adaptive MPI [14], are designed not only to implement the MPI Standard as defined by the MPI Forum, but to add additional features such as fault tolerance, process migration, load balancing, and more.

In this dissertation, we use MPI as a basis for all work. It provides the communication mechanism which we harden by extending the definition of the MPI Standard to include fault tolerance. In the meantime, we gain the experience of the research that has gone into providing a complete and optimized set of communication tools.

2.2.1 Other Communication Libraries

PVM

While MPI has become the most popular communication library, it is not the only mechanism available. PVM (Parallel Virtual Machine) [57] existed before MPI and provided a view of the machine as a large "virtual machine" abstracting the underlying hardware and network topologies. In this sense, PVM was designed to provide simultaneous, large-scale computing across a range of machines while presenting a simple, easy to understand interface for the programmer. It accomplished this model by providing tasks as the basis for a PVM application. Each task is deployed onto a host from a pool of available hosts, and hosts can be dynamically added and removed from the pool at runtime. PVM also used message passing to perform its communication, but the task system also facilitated other features, including fault tolerance and heterogeneity. By allowing hosts to enter and leave the pool dynamically, process failures did not have to cause the entire application to fail. Instead, the failed processes were excluded from the host pool and more hosts were selected to replace them.

Charm++

The idea of portable tasks as the fundamental piece of an application was expanded on in Charm++ [44]. Developed at the University of Illinois at Urbana-Champagne, Charm++ is a programming language based on C++. Applications are again broken down into tasks, called *chares*, and virtually mapped onto processes with an "intelligent runtime" system which dynamically evaluates all running applications to put chares in the most appropriate location on the system. Another feature of the intelligent runtime is that it can manage the deployment of chares even after they have started execution. When the runtime detects that the deployment of chares is imbalanced among the nodes, it can automatically migrate processes on-the-fly to better balance the computation load and increase performance. This also provides fault tolerance at a fundamental level as chares can be replaced on the fly without restarting the entire application. Charm++ was later used to create an MPI implementation called Adaptive MPI [14] which included many of the features from Charm++, such as migratable processes and fault tolerance support.

Partitioned Global Address Space

Partitioned Global Address Space (PGAS) languages strive to simplify programming techniques by allowing applications to refer to data arrays as if stored locally, while automatically managing data distribution within the library implementation. A number of programming libraries and languages strive to provide this capability.

Chapel [20] (the Cascade High Productivity Language) is a parallel programming language developed collaboratively by Cray, academia, industry, and scientific computing centers. Fundamentally, Chapel abstracts away much of the challenge of high performance computing. It is specifically designed with four main goals: multithreading, locality-awareness, object-orientation, and generic programming. These goals allow the user to achieve high performance while simplifying programmability. Despite its goals, Chapel has still not reached the level of adoption of MPI.

Unified Parallel C (UPC) [22] extends the ISO C 99 Standard by adding, in addition to PGAS abilities, synchronization primitives and simpler memory management. Like Charm++, UPC can be implemented on top of MPI, using MPI as the communication mechanism while simplifying programmability through its language extensions. High Performance Fortran (HPF) [52] tries to provide many of the same capabilities, but with the FORTRAN 90 specification instead of C. Global Arrays (GA) [1, 48] is similar, but attempts to provide more portability and interoperability with other parallel programming libraries, such as MPI by providing a library interface, rather than an entirely new language.

2.3 Types of Fault Tolerance

Now that the programming model for our application has been established, we will examine some other fault tolerance solutions which have been produced. The types of fault tolerance available to applications can be overwhelmingly numerous. From system-level to user-level, automatic to user-involved, making the correct decision about which type of fault tolerance to use in an application is important not only from a performance perspective, but also to enhance programming productivity by choosing a fault tolerance solution that is understandable and most appropriate to the environment in which it is being used. Here we describe each of these types of fault tolerance to give context to how the work presented in this dissertation can be categorized.

2.3.1 System-Level vs. User-Level Fault Tolerance

Fault tolerance comes in two broad categories: system-level and user-level. System-level fault tolerance includes checkpoint/restart systems that capture the entire range of application memory automatically. They can be executed automatically by some entity other than the user application and are designed to be the simplest to use, though often with a higher cost. Because they capture all of the data used by the application, whether important or not, they can be inefficient when storing and retrieving large amounts of unnecessary data. Examples of system-level fault tolerance include early checkpoint/restart libraries which did not include user selectable checkpointing.

Alternatively, user-level fault tolerance trades simplified, automated fault tolerance for a more efficient, but less automated style. The application determines which parts of its data are most important and protects only those parts, allowing the remainder of the data to be reconstructed using other methods. The application can also control features such as the time and frequency within the execution where checkpoints would be least costly and most effective. Because of this selectiveness, user-level fault tolerance tends to perform better than system-level, but can be more challenging to use.

2.3.2 Checkpoint/Restart

The most prevalent form of automatic fault tolerance is checkpoint/restart. It is the simplest form of fault tolerance to explain and understand as it so closely resembles an action which all computer users employ constantly, saving the state of an application to disk. While different implementations provide this functionality in different ways, the overarching functionality of checkpoint/restart is to save some subset of the state of an application to a location which will be available at a later time, and retrieved and restarted to continue the application.

Global State

All checkpoint/restart libraries rely on the theory of the work done in Chandy and Lamport's global state research [24]. In this work, they define a method of capturing a global state of a running computation by monitoring the status of communication channels between nodes. They describe the idea of the work using an photography analogy:

> The state-detection algorithm plays the role of a group of photographers observing a panoramic, dynamic scene, such as a sky filled with migrating birds - a scene so vast that it cannot be captured by a single photograph. The photographers must take several snapshots and piece the snapshots together to form a picture of the overall scene. The snapshots cannot all be taken at precisely the same instant because of synchronization problems. Furthermore, the photographers should not disturb the process that is being photographed; for instance, they cannot get all the birds in the heavens to remain motionless while the photographs are taken. Yet, the composite picture should be meaningful.

From this work, we discover two methods of performing checkpoints. The first, simpler method is to stop the computation and record the global state of the algorithm on all nodes at once. This method is simple to understand, but incurs a high overhead as all computation must stop and the checkpoint operation must be completed before the algorithm may continue. This is analogous to the "stop the heavens" solution described above. The second method is more complex, but does not require the running algorithm to be disturbed. Instead the local state of a process is stored and the messages sent between processes are cached using an algorithm similar to that proposed by Chandy and Lamport.

libckpt

One of the first available checkpointing libraries was libchkpt [49], a hybrid of system and user-level checkpointing. Like BLCR, though developed before it, libckpt could provide automatic checkpointing in a way that was virtually invisible to the application (the library required modifying one line of code). However, where libckpt differentiated itself from other checkpointing libraries was with an array of other improvements. Incremental checkpointing improved the checkpoint write times and storage requirements by saving only the difference between the current checkpoint and the previous one. Forked checkpointing removed the sequential nature of checkpointing, where the application execution was interrupted to create the checkpoint and resumed upon completion, instead replacing it with a system where a child process is created by the checkpointing library to perform the checkpoint. libckpt also included a relatively new feature at the time to write checkpoints as directed by the application itself. The goal was to minimize the size and frequency of the checkpointing operation. The mechanisms introduced were memory exclusion, where certain portions of memory for which protection was no longer required could be excluded from the checkpoint, and synchronous checkpointing, where the application could take a checkpoint at a specific time in the code where it would be most advantageous. The pairing of these two operations could create a minimal checkpoint, both in terms of time and size.

Libckp

Libckp [61], developed at AT&T Bell Labs, is another user-level checkpoint library that differentiated itself from others by including a more robust file checkpointing system and a design specifically tailored to fault tolerance. First, libckp could not only recover access to files when a process is restored from a checkpoint, but also the status of the files at checkpoint time could be restored to ensure that the environment is consistent with the state in which the checkpoint was made. In addition, libckp could also roll back running applications to a previous state. Other libraries were designed to restart an application from a checkpoint or migrate processes to a new location after a checkpoint was made. libckp could roll any remaining processes back to a consistent state after a failure to reduce the overhead of recovery. This was an important step to facilitating real fault tolerance inside the application.

Condor

Condor [46, 47] (now called HTCondor), is a suite of tools developed to reclaim unused computation cycles on a network of workstations. As part of the suite, a checkpoint/restart system was needed. The C/R capabilities would allow an application to move from one workstation to another using process migration. As users would start to use a workstation, the application would be checkpointed by Condor and migrated to another unused workstation where the application would continue. Condor, as with many other user-level implementations, does have some limitations where data from some sources cannot be saved due to inaccessibility outside the system kernel. Condor is still in development today and can be acquired from its website.

BLCR

System-level checkpoint/restart systems, such as the Berkeley Lab Checkpoint/Restart (BLCR) [33] library, provide the most complete form of C/R by integrating with the system kernel to capture all available information about a process, from its process and session IDs to the entire contents of its memory. It is able to store all of this information in a way that can later be completely replicated to bring an application to the exact point of execution where the checkpoint was captured. BLCR describes the goals of such functionality to provide not only fault-tolerance, but also gang scheduling, where short interactive jobs can be scheduled on hardware during working hours and longer-running, non-interactive jobs can be rescheduled at night, when the speed of job interaction is not as critical. Also, such system-level checkpointing facilitates job migration between nodes when one node is underperforming or before a failure occurs. The management system can automatically reload the application on a new machine without the application modifying any code or needing to be aware of the migration at all.

Asynchronous Checkpointing

Asynchronous checkpointing is similar to the more familiar and traditional synchronous checkpointing when done in user-space. In both models, individual processes are responsible for saving their own data to disk, however the difference comes in the coordination of such checkpoints. For asynchronous checkpointing, all processes do not write the checkpoint at the same time. Instead, they checkpoint their local data at a time which makes sense and then log messages the are sent between the checkpoints. When rollback is necessary, the checkpoint is reloaded and the messages are replayed to bring the recovering processes back to the same point as the remaining processes. Many implementations of asynchronous checkpointing exist [17, 51] and will be detailed later.

2.3.3 Migration

Process migration is another automatic solution which takes advantage of rollback recovery techniques in a different way. Classically, after a failure, the job is restarted using largely the same physical machines, but substituting the failed machines for some which are still running. In some instances these issues can be avoided if sufficiently accurate failure predictors are available. In these scenarios, processes can checkpoint automatically and move from a suspected node to a node where the failure probability is lower. Much of the work in this field originated when designing NoW clusters where machine availability was based on the idleness of the workstations [61, 46, 33], however with newer failure predictors, this technique is now being deployed on large-scale HPC machines to combat failures through libraries such as Adaptive MPI.

2.3.4 Replication

Replication has recently been proposed as a solution to the increasing cost of checkpointing, both synchronous and asynchronous [36]. The idea of replication is that most applications do not use the entire machine size on the machines on which they are run. To take advantage of the "wasted" space of the machine, multiple copies of the application are run simultaneously. If a process failure occurs, one of the replicant processes is wired into the original version of the application and the computation can continue without the rollback requirement. This solution has been demonstrated to have merit for some types of machines, especially those where the system utilization is not greater than 50%. However, for typical HPC deployments, where a machine is not utilized for capability jobs, but for capacity, where smaller jobs fill out the time on the system and very few jobs take advantage of the entire system size, this technique crowds out the other jobs by imposing an overhead of at least 100% of the original job size (maybe more if more than one replicant is used). Bosilca et al [16] performed a study to demonstrate the overhead of various fault tolerant techniques which demonstrates the tradeoff points between rollback-recovery and replication. In addition to demonstrating these overheads, this paper also points to the need for new fault tolerance techniques for capability applications of the future.

2.3.5 Algorithm Based Fault Tolerance

All of the above automatic fault tolerant solutions have been productive on existing systems where bottlenecks such as I/O bandwidth did not yet cause issues. Today, new fault tolerance techniques are needed to lower the overhead of resilience and ensure that HP applications will continue to be productive in the future.

Algorithmic Based Fault Tolerance (ABFT) began as a field of study to resolve silent errors in linear algebra problems. Since that time, it has expanded to include diskless checkpointing and more sophisticated techniques. This section will examine this progression.

Silent Error Detection

While many times errors are thought of as problems which cause catastrophic machine failure of some kind, this is not always the case. Some failures, such as a bit flip due to radiation, do not create an easily detectible failure, rather they introduce a small distortion in the contents of memory. These types of failures must be detected by the algorithm itself to ensure that a correct answer has been reached. The need for this sort of failure detection spawned ABFT [40]. Huang and Abraham introduced a new method of evaluating the results of linear algebra computations to ensure accurate results. From their work, a new field of study emerged.

Diskless Checkpointing

To support ABFT, a new set of tools was necessary. Though checkpoint/restart had existed previously, it usually required that the entire application be stopped and restarted by reloading a checkpoint from disk. This could be a very expensive operation as the bandwidth to the stable storage could be a bottleneck, and often, most of the processes were still functioning correctly and did not require a restart. To solve this problem, Plank, Kim, and Dongarra [50] created a new form of checkpointing called Diskless Checkpointing. Rather than writing the checkpoints to stable storage to be expensively re-read when a failure occurred, often back to the same location as the original copy, the checkpoints would be stored directly in the memory of the remaining processes. Supported by their previous work [49], this facilitated fast retrieval and no longer required all of the processes to restart, regardless of which processor suffered the failure.

As MPI techniques for recovery became more sophisticated (see Section 2.4 for more details), new algorithms were developed to take advantage of the new features using diskless checkpointing. One of the first to do this was a Preconditioned Conjugate Gradient solver [25]. This algorithm used weighted checksums and additional MPI processes to calculate and store checksums which could be retrieved after a process failure. FT-MPI provided the mechanism to replace failed processes and allowed the application to continue communication. This technique expanded to other applications, including the QR Factorization used as an example in this work.

2.3.6 Transactional Fault Tolerance

Transactional fault tolerance has been in existence for decades. It began as a way of ensuring data consistency within distributed databases [13]. Each operation submitted to the database was either applied completely and successfully, or the database was rolled back to a state before the operation was attempted. By performing updates in this atomic fashion, the database was protected from corruption in the case where the operation failed. Later, as concurrency became more popular in computing, transactional memory was introduced [39] to assist the programmer when attempting to ensure that multiple concurrently running processes did not attempt to write to the same piece of memory at the same time. The ideas of transactions are currently moving into HPC, including preliminary discussions of transactional fault tolerance in the MPI Standard.

2.4 MPI Level Fault Tolerance

As research in fault tolerance matured and developers started to integrate it into their codes, new efforts began to bring the ideas of fault tolerance into the most popular distributed communication library, MPI. Though fault tolerance has never been an official part of the MPI Standard, work toward achieving fault tolerance within MPI has continued for many years. In this section, we will examine some of the efforts made to extend MPI.

FT-MPI & HARNESS

FT-MPI [34] has been one of the most successful (in terms of number of users) implementations of fault tolerance within the MPI stack, to date. FT-MPI provided a number of options for users to recover from failures. All of the options provided mostly automatic recovery within the MPI library itself, minimizing the impact of fault tolerance on existing codes. To trigger the recovery, the user only needed to call a new communicator creation function (such as `MPI_COMM_DUP` or `MPI_COMM_CREATE`) and the MPI implementation would automatically construct the new communicator in such a way that it would be functional for MPI communication.

The first recovery mode option is SHRINK. In this recovery mode, failed processes are removed from any new communicator, and the communicator returned from the creation function will compress the remaining processes to create a monotonically increasing set of ranks. For some applications, this could cause problems due to calculations that depend on a consistent value for the local rank.

The second recovery mode option is BLANK. This mode is similar to SHRINK in that all failed processes are removed from the new communicators. However, rather than compressing the remaining processes, the failure mode replaces them with an invalid process. Any attempts to communicate with the invalid ranks will cause an error. This failure mode provides the opportunity to replace invalid ranks with new processes at a later time, but maintains the ranks of existing processes where such a value is important for the computation.

The third and most well-supported recovery mode is REBUILD. This recovery mode automatically creates new processes to replace any processes which have failed. The goal of this recovery mode is to support applications where a consistent number of processes with consistently numbered ranks can be guaranteed. New processes are automatically restarted with the same command line parameters as the original processes; however, they are not automatically reinserted into all of the subcommunicators of the processes which they replace. Any communicators other than MPI_COMM_WORLD must be reconstructed manually.

When FT-MPI was first written, it was actually built on top of PVM due to the lack of an appropriate MPI runtime. Later, the HARNESS runtime [35], originally implemented in Java, was rewritten in C and adopted as the foundation for FT-MPI. This runtime provided key functionality such as the ability to create new processes, monitor their health, and track the status of all processes from any node.

FT-MPI was a successful project in the sense that it started an important area of research into fault tolerance included in the MPI specification, though it was never adopted into the MPI Standard itself. Because of the lack of standardization, its availability and financial support was eventually crippled and the project is no longer maintained.

MPI with Checkpoint/Restart

Alongside the efforts described in Section 2.3.2 have been similar efforts to integrate coordinated Checkpoint/Restart mechanisms into the MPI implementations themselves. One of the first attempts to accomplish this involved BLCR and LAM/MPI [54]. A team of researchers from Indiana University and Lawrence Berkeley National Laboratory integrated BLCR's kernel-level process checkpointing with the LAM/MPI implementation using coordinated checkpointing to automatically preserve an application after failure. These checkpoints can either be triggered transparently by the MPI implementation or manually by the user. Upon failure, the user can restart the application from a saved process context using the BLCR utility `cr_restart`.

In addition to the work to integrate checkpoint/restart functionality into the MPI library itself, there has been work to improve the performance of such checkpointing operations, both when writing checkpoints and later reading them back. In [60], the authors claim the the primary overhead of checkpoint/restart implementations is the bottleneck of accessing the data storage device with many nodes at once. To resolve this issue, they proposed an algorithm using LAM/MPI which would instead distribute the checkpoints to a number of nodes. This would improve the performance because the data was no longer being sent to the same point, but distributed among the entire system. In addition, the resiliency was improved because multiple copies of the data ensured that even if one node experienced a fault, the other nodes would still be available to provide the data on recovery.

Later, LAM/MPI was merged with many other MPI implementations to form Open MPI and the work on checkpoint/restart systems was revived in this new context [43]. In addition to supporting BLCR, the new work focused on providing a framework to allow other forms of checkpoint/restart to function as well, including asynchronous checkpoints. The new system had five primary tasks on which it focused: a snapshot coordinator to initialize, monitor, and aggregate checkpoint data, file management tools to ensure that snapshot data is available on the correct nodes at the correct time, a distributed checkpoint/restart protocol to coordinate the checkpointing itself, implemented at the MPI layer to provide the most support for the intended system, a local checkpoint/restart system to actually perform the checkpointing operation on the nodes, and a notification mechanism for the MPI library itself to preserve its own state for the checkpointing operation. This system was successful in its implementation, integrating tightly with all components of the Open MPI implementation.

MPICH-V

MPICH-V [15] was designed to be an MPI implementation providing automatic, transparent fault tolerance via asynchronous checkpointing and message logging. It accomplished this by starting with a well-known implementation, MPICH, and adding the needed mechanisms. Checkpoints were performed locally using the Condor Stand-alone Checkpointing Library (CSCL), using the forked checkpointing method described earlier, and sent to a checkpoint server where they are stored until needed upon a node failure. Similarly, to achieve a total ordering of messages and maintain a complete record, messages are routed through Channel Memory (CM) nodes. These nodes maintain a record of the messages and provide them to restarted processes when necessary. In addition to its fault tolerance features, MPICH-V was also designed to create grids of workstations, including handling local firewalls when routing messages.

MPI/FT

Another MPI implementation, MPI/FT [12], tried to create a hybrid of available fault tolerance solutions to support different styles of applications. It targeted two specific models. The first model was a master-worker style application where a single master process communicates directly with worker processes but does not use collective operations. For this type of application, the MPI library would notify only the master process of failures and automatically relaunch failed workers and repair the communication channels of `MPI_COMM_WORLD` on the master process. Checkpointing was unnecessary for this type of application as the worker processes did not have critical data which could be easily recalculated. The second model of application was the more traditional, Single Program Multiple Data (SPMD) application, where any process may communicate with any other process. For this type of application, the MPI library expected the application to perform synchronous loops on all processes where communication was clustered at the beginning and end of each loop. This model did include checkpointing which was coordinated by the process at rank 0. When a failure occurred, the failed process was replaced from the checkpoint and `MPI_COMM_WORLD` was also repaired to include the replaced process on all ranks. Recovery was not handled automatically, but done via new API calls to coordinate recovery, query the system about process status, and other interfaces. While MPI/FT provided an interesting solution for specific models of applications, it was not a complete solution for applications which required more exotic executions.

Egida

Egida [51] is a toolkit, integrated with MPICH, designed to provide fault tolerance for non mission-critical applications that run on NoW clusters rather than large scale machines. It provides transparent recovery through log-based rollback recovery. To perform all the tasks necessary to accomplish this, it uses a series of modular building block kernels to support a monolithic event handler. As events are ordered from the software API, failure detector, network monitor, or timers, the event handler executes a grammar to determine the appropriate course of action, and activates the necessary kernel to perform the action.

Starfish

Starfish [10] is another fault tolerance solution targeting NoW clusters. The main contribution to differentiate Starfish from other solutions is that it was one of the first such implementations to support MPI-2 dynamic processes. It is constructed by deploying Starfish daemons on each machine which function to deploy and manage application processes, including MPI processes. The daemons are also responsible for the deployment of the fault tolerance solution. Starfish is designed to use checkpointing (either synchronous or asynchronous) and can interchangeably deploy new checkpoint/restart systems as they are implemented. Without modification, MPI applications integrating with Starfish can only use transparent, system checkpoints. To use additional functionality, such as user-initiated checkpointing and dynamic process reconfiguration, the application must be modified to use Starfish specific up-calls and down-calls.

DejaVu

DejaVu [53] is a transparent checkpoint/restart system incorporated with MPI. It provides low-overhead checkpointing and message logging to ensure resilience across any number of failures. By bringing the checkpointing into the library, it can provide the portability not available from traditional system or user- level checkpointing solutions. Also, by including message logging, DejaVu can support process migration in the event that a process cannot be relaunched on its original node.

Proactive Fault Tolerance

Not all fault tolerance techniques require that the failure has already occurred before reacting. In [23], the authors demonstrate a scheme where the MPI implementation can take advantage of highly accurate failure predictors, using existing hardware monitors such as temperature sensors, to predict when a failure might occur and preemptively move the running task to another node. This work uses Charm++ and Adaptive MPI as a basis for its work to take advantage of its built-in task migration capabilities. When the failure prediction techniques cannot predict a failure, the implementation relies on traditional checkpointing techniques to recover from failures.

2.4.1 Open MPI

In this dissertation, we base our work on the Open MPI implementation of the MPI Standard. Open MPI [8, 38] has been designed to be a collaboration between industry, academic, and research partners to design an MPI implementation which facilitates both high performance communications, but also research involving future technologies which could be involved with MPI. Open MPI has a modular design which is intended to facilitate easy replacement of specific parts of the implementation in order to develop new modules. This has lead to a large number of systems which are supported by the implementation and thus, a large user base. Recently, Open MPI was used as the basis for the MPI implementation which led to the Japanese K-Computer becoming the fastest machine in the world for the June 2011 Top500 list [3].

2.5 Conclusions

The thread that unites all of these fault tolerance solutions (other than FT-MPI) is that none is designed to allow MPI applications to continue communication *after* a process failure. They allow recovery by redeploying the application after a failure and restoring from a previous state. In this sense, they are not solutions to fix MPI itself, but to fix the applications running on it. In the remaining chapters, we will demonstrate a solution using both existing MPI implementations to repair applications after a failure, and a new MPI proposal which provides continuous communication across process failures.

Chapter 3

Design Goals

After evaluating the features, strengths and weaknesses of the previous research in fault tolerance, four main goals for a successful fault tolerant communication library emerge. While not all libraries will fulfill all of these goals, for an MPI library to be successful at supporting a wide variety of fault tolerant paradigms, these are foundational principles which should be considered during the design.

3.1 Flexibility

A successful fault tolerant library must provide the flexibility to support multiple consistency and recovery techniques. For example, a Monte-Carlo master-worker application may not require complex recovery after a process failure is detected by the master process. Rather, the failed worker process can safely be ignored. If an MPI implementation attempts to perform some method of automatic recovery, it would not only introduce a high recovery cost to an application which does not require it, but it would also require the application to change its behavior in order to support the type of fault tolerance mandated by the library. This is not the most flexible approach to fault tolerance and therefore limits its usefulness as part of a communication standard. Rather, the standard should provide the minimum level of recovery, only enough to allow further communication, and then allow the application to choose what direction subsequent recovery should take.

3.2 Resilience

Resilience refers not only to the ability of the MPI application to survive failures, but also to recover into a consistent state from which the execution can be resumed. This manifests most profoundly in the effort to ensure that an MPI operation cannot stall indefinitely as a consequence of a failure. If an operation never returns, the application can take no part in the recovery, and fault tolerance is impossible. All operations which perform communication must return descriptive error codes to inform the application of any unexpected behavior which occurred while the library was executing. As long as some processes in the application are informed of a failure, they can initiate recovery actions. In addition to being deadlock-free, the library must also provide mechanisms to alert other processes to failure when necessary. These mechanisms could be automatic within the library or manual via an external construct.

3.3 Performance

The performance impact of any fault tolerance additions to an MPI communication library must be minimal when outside of the recovery path. Internal recovery should be triggered only when necessary and normal failure monitoring actions should take place out of the performance critical path. As mentioned in Section 3.2, not only should the failure-free operations introduce insignificant levels of overhead, but recovery operations should also be fast. Many automatic fault tolerance techniques exhibit poor performance as they require universal participation in recovery after a failure. Rather than imposing such global knowledge on the system, a minimal, local knowledge shows much more promise for high performance. When alerted to a failure, if it is necessary to inform other processes, appropriate constructs should be called by the application, not the library, to ensure that only necessary levels of recovery are executed.

3.4 Productivity

The last goal is harder to measure empirically, but is nonetheless critical in the design of a fault tolerant MPI library. An enormous number of legacy MPI codes already exist which do not support fault tolerance and would not benefit from its support if it were to be implemented. To that end, any new fault tolerance additions to the MPI Standard must not require changes from such legacy applications. This means that the behavior of existing MPI operations should not change without a severe need.

In addition, the fault tolerance constructs should be minimal both in terms of quantity and complexity. By providing the minimal set of changes to MPI, the chances of the library being used increase and the time required to adopt the library decreases.

When designing a minimal set of changes to supply fault tolerance, some convenience functions which might increase programmability will be left out. This does not prohibit such functions from existing. Instead, these functions may be provided as an external library built on the foundation of a minimal standard. These external libraries are not limited only to convenience functions. They can also introduce complex recovery mechanisms not found in a standardized document.

Chapter 4

Checkpoint-on-Failure

As a first attempt to meet the goals set out in Chapter 3, we evaluated the feasibility of implementing fault tolerance in the context of the current MPI Standard (version 3.0 [59]), using only the mechanisms available as it is currently written. This chapter details that effort and demonstrates an application that can function under such constraints.

4.1 Existing Error Handling in MPI

The existing MPI Standard provides minimal support for fault tolerance. Section 2.8 states in the first paragraph:

> MPI does not provide mechanisms for dealing with failures in the communication system. [. . .] Whenever possible, such failures will be reflected as errors in the relevant communication call. Similarly, MPI itself provides no mechanisms for handling processor failures.

Failures, be they due to a broken link or a dead process, are considered resource errors. Later, in the same section:

> This document does not specify the state of a computation after an erroneous MPI call has occurred. The desired behavior is that a relevant

> error code be returned, and the effect of the error be localized to the greatest possible extent.

So, in the existing standard, process failures are treated as errors, and therefore the behavior of the MPI library is undefined. However, the standard does provide guidance for implementations to be considered “high quality”. The second excerpt hints at such behavior by suggesting that the library attempt to localize the impact of the error and inform the application of the failure. Unfortunately, most of the implementations of the MPI Standard have implemented process failures as unrecoverable errors, and the processes of the application are most often killed by the runtime system when a failure is detected on any of them, leaving no opportunity for the user to mitigate the impact of failures.

In addition to this limited definition of the behavior of the library after a process failure, MPI also defines a construct called an `MPI_ERRHANDLER`. These are designed to be triggered when a high quality implementation of MPI detects a failure of some kind. The `MPI_ERRHANDLER` is attached to an MPI Communicator object and includes a callback function which is executed by the library. MPI provides two built-in error handlers, `MPI_ERRORS_ARE_FATAL` and `MPI_ERRORS_RETURN`. `MPI_ERRORS_ARE_FATAL` is the default error handler, and when MPI detects a failure, it automatically aborts the entire MPI application without the possibility of recovery or cleanup. `MPI_ERRORS_RETURN` provides more functionality by attempting to return control to the application after a failure. MPI is no longer usable for communication, but the application can perform actions to clean up the system before exiting. Custom error handlers provide the most flexibility. Their callback function can perform last second operations as the MPI library becomes unusable.

4.2 The Checkpoint-on-Failure Protocol

Based on the capabilities of the current version of the MPI Standard, we designed a new approach for supporting ABFT applications, called Checkpoint-on-Failure (CoF).

Table 4.1: The Checkpoint-on-Failure Protocol

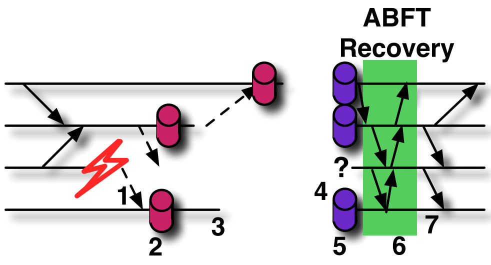
1. MPI returns an error on surviving processes 2. Surviving processes checkpoint 3. Surviving processes exit 4. A new MPI application is started 5. Processes load from checkpoint (if any) 6. Processes enter ABFT dataset recovery 7. Application resumes	

Table 4.1 presents the steps involved in the CoF method. In the figure, horizontal lines represent the execution of processes in two consecutive MPI applications. When a failure eliminates a process, other processes in the application are notified and regain control from ongoing MPI calls (1). Surviving processes should assume the MPI library is dysfunctional and not continue to use MPI operations (in particular, they do not yet undergo ABFT recovery). Instead, they checkpoint their current state independently (2) and abort (3). If any processes were not initially alerted to the failure, they will eventually be notified after the cascading calls to `MPI_ABORT` reach one of their neighbors. When all processes have exited, the job is usually terminated, but the user (or a managing script, batch scheduler, runtime support system, etc.) can launch a new MPI application (4), which reloads processes from checkpoint (5). In the new application, the MPI library is functional and communications are possible; the ABFT recovery procedure is called to restore the data of the process(es) that could not be restarted from checkpoint (6). When the global state has been repaired by the ABFT procedure, the application is ready to resume normal execution (7). If another failure hits the system during the recovery, the local states are not updated, and the relaunch starts from the beginning. If another failure hits the system after the ABFT recovery, the entire procedure is followed to handle it.

CoF is most directly comparable to the existing method of periodic checkpointing. Compared to periodic checkpointing, in CoF, a process pays the cost of creating a checkpoint only when a failure, or multiple simultaneous failures, have happened, hence it creates an optimal number of checkpoints during the run (and no checkpoint overhead on failure-free executions). Moreover, in periodic checkpointing, a process is protected only when its checkpoint is stored on safe, remote storage, while in CoF, local checkpoints are sufficient: the forward recovery algorithm reconstructs datasets of processes which cannot restart from a checkpoint. Of course, CoF also exhibits the same overhead as the standard ABFT approach: the application might need to do extra computation, even in the absence of failures, to maintain internal redundancy (whose degree varies with the maximum number of simultaneous failures) used to recover data damaged by failures. However, ABFT techniques often demonstrate excellent scalability; for example, the overhead on failure-free execution of the ABFT QR operation (used as an example in Section 4.5) is inversely proportional to the number of processes.

4.3 MPI Requirements to support CoF

To support CoF, some demands are made of the underlying MPI implementation.

Returning Control After Failures: Many MPI implementations not only choose `MPI_ERRORS_ARE_FATAL` as the default MPI error handler, but also either do not implement the ability to choose another error handler, or provide other error handling mechanisms which supercede the MPI error handlers. For CoF to be functional, the MPI implementation must provide a functional error handler mechanism, including `MPI_ERRORS_RETURN` as well as custom error handlers defined by the application. In addition, the MPI implementation must guarantee that after a failure, it returns control to the application by invoking the error handler. It does not need to provide a perfect failure detector [37] where all processes are immediately notified of failures (indeed, that is often the wrong type of failure notification due to its high overhead cost), but it must never deadlock because of a failure, preventing the application from regaining control and performing recovery mechanisms, defined as an eventually perfect failure detector.

Termination After Checkpointing: The application must be able to reliably ensure that all other processes are notified of the failure after any process detects it. This can be through a user-controllable mechanism, such as exiting without calling `MPI_FINALIZE` or by invoking `MPI_ABORT`, but if the failure is not propagated, then some processes will take a recovery execution path while others attempt to continue normal execution and eventually reach a deadlock scenario.

4.4 Open MPI Implementation

Open MPI is an MPI 2.2 compliant implementation of the MPI standard. Architecturally, it is divided into two main levels: the runtime (ORTE) and the MPI library (OMPI). As with many MPI implementations, the default behavior of the library is to abort upon process failure. This policy was implemented deeply at the runtime layer, preventing the OMPI layer from making any policy decisions on the status of the library after a failure. To correct this, major changes needed to be implemented in ORTE.

4.4.1 Resilient Runtime

The main contribution of the Open MPI runtime is to provide process management. This includes creating new processes at the beginning of an MPI job, cleaning up processes at the end of the job, and spawning new processes within the job if requested by the application. To accomplish this task, ORTE includes an out-of-band (OOB) communication mechanism which allows the ORTE layers to communicate amongst each other without impacting the performance of the high performance network. When a node failure occurs, not only is the application's communication ability impacted, but also the runtime. ORTE needs to be able to react and repair the

OOB communication topology to route around failures and allow itself to continue process management. For some communication topologies, such as a star where all processes are directly linked to the head node, this is a trivial operation and only requires excluding any failed processes from the routing tables. For more elaborate topologies, such as a binomial tree, the self-healing operations are more complex, requiring each node to recompute the tree around it to repair any links. If a parent process in the tree fails, the healing process needs to make a new link to the next alive process traveling up the tree. If a child process fails, the same needs to happen traveling downward. In this way, the tree will always remain connected among alive processes. However, it is not guaranteed, and indeed unlikely, that the tree will remain balanced. It was determined that this was not a critical requirement of the OOB as it is not used as a high performance messaging layer in Open MPI, only as a communication mechanism for process management.

4.4.2 Failure Notification

In addition to providing a self-healing OOB communication mechanism to facilitate CoF, ORTE also needed to provide basic failure notification. To track the status of failures, an incarnation number has been added to the ORTE process names. When a failure is detected, the name of the failed process (including its incarnation number) is broadcast over the OOB topology. The incarnation number provides a mechanism to determine which process failures are already known and which are not, preventing duplicate recoveries for the same process failure. It also prevents a transient process failure from causing confusion among the processes in the OOB by ensuring that all processes know with which incarnation of a particular process they should be communicating. To propagate this knowledge, ORTE processes monitor the health of their neighbors in the OOB topology. When a failure is detected, the processes around the failed process perform the route healing described in 4.4.1 followed by a reliable broadcast algorithm which informs all processes in the application of the failure. This algorithm has a low probability of creating a bifurcation of the routing topology. Indeed, in the provided OOB topologies, this algorithm will never produce a bifurcation. On each node, when the ORTE layer is notified of process failure, it forwards the information to the OMPI layer, which has been modified to invoke the appropriate MPI error handler, as determined by the user.

4.5 Example: QR-Factorization using CoF

This section illustrates the usefulness of CoF by demonstrating its applicability to a widely used class of algorithms: dense linear factorizations. The linear algebra algorithm modification performed in this section was done by Peng Du building on the CoF library we implemented. The QR factorization is a cornerstone in many applications, including solving $Ax = b$ when matrices are ill-conditioned, computing eigenvalues, least square problems, or solving sparse systems through the GMRES iterative method. For an $M \times N$ matrix A, the QR factorization produces Q and R, such that $A = QR$ and Q is an $M \times M$ orthogonal matrix and R is an $M \times N$ upper triangular matrix. The most commonly used implementation of the QR algorithm on a distributed memory machine comes from the ScaLAPACK linear algebra library [31], based on the block QR algorithm. It uses a 2D block-cyclic distribution for load balancing, and is rich in level 3 BLAS [28] operations, thereby achieving high performance.

4.5.1 ABFT QR Factorization

In previous work [32], the QR factorization algorithm written for ScaLAPACK was modified to include ABFT techniques, leveraging FT-MPI as the platform to maintain MPI communication after a failure. To ensure that the data in both the left (Q) and right (R) factors is protected from fail-stop errors during execution, a technique called Reverse Neighboring Checksum Storage is used. For each group of (Q) processes, a

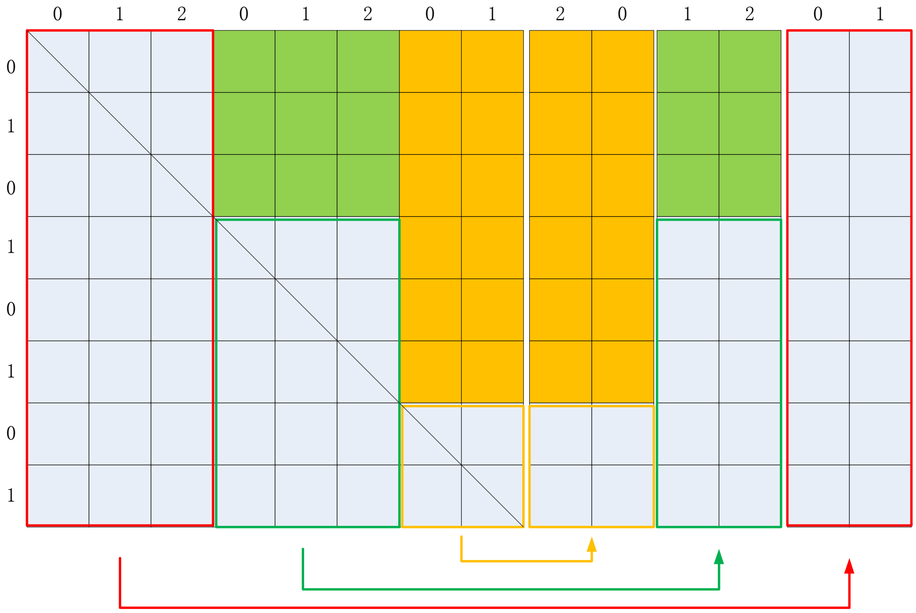


Figure 4.1: Pattern to store checksums to prevent data loss in the event of multiple failures. Figure borrowed from [32]

.

checksum and a duplicate are calculated and stored at the end of the matrix as shown in Figure 4.1. These checksums are automatically updated as the algorithm is executed. The matrix-matrix multiplication which updates the right side of the original matrix will also update the values of these checksums. Because of this property, the more expensive checksum operation is absorbed by the already executing DGEMM kernel.

When a process failure is detected, all remaining processes are alerted of the location of the failure by FT-MPI, which creates a replacement process in the same coordinates in the $P \times Q$ block-cyclic distribution as the failed process. To restore any missing checksums, the value is simply copied from a duplicate. To restore the missing data blocks within the right factor of the matrix, a reduction operation calculates the value of the missing data by subtracting the remaining data values from the checksum. The value of the left factor is also stored in a checksum at the bottom of the matrix.

Those values are either recovered from the checksum similarly to the right factor, or the most recent panel is recomputed.

While this algorithm was successful using FT-MPI, as previously stated, FT-MPI does not remain a viable candidate for MPI fault tolerance in the future. The algorithm was ported to a more compliant version of MPI, Checkpoint-on-Failure, to demonstrate its feasibility on existing systems.

4.5.2 Checkpoint-on-Failure QR

Checkpoint Procedure: Compared to a regular fault tolerance tool, CoF is not a standard checkpointing procedure. Where system-level checkpoints save the contents of large sections of memory, whether the data is still useful or not, CoF applications should only checkpoint the most vital pieces of data that are either required for an application to resume, or are prohibitively expensive to recalculate at recovery time. This means that codes should refactor their existing checkpointing functions to save less data and store it in a different location (depending on the type of application and execution environment). For CoF-QR, the checkpointing function saves the local values of the matrices and the loop indices necessary to restart. All other data critical to the application can be regenerated quickly from these most important pieces.

State Restoration: ScaLAPACK programs have deep call stacks, including functions from several software packages, such as PBLAS [26], BLACS [29, 30], LAPACK [11] and BLAS [27]. In the previously existing FT-MPI version of the QR algorithm, regardless of when the failure was detected, the current iteration of the algorithm needed to be completed before processing the recovery procedure. This would ensure an identical call stack on every process and that all processes had updated their checksums completely. For the new CoF version of QR, failure must interrupt the algorithm immediately, not completing the current iteration, because the MPI library can no longer support the communication necessary to calculate the most up to date checksums. While this has the potential to cause divergent call stacks among the processes, because failure notification happens only in MPI and the lower level procedures (BLAS, LAPACK, etc.) do not perform communication, the data remains uncorrupted by failures.

To resolve the call stack issue, when restarted, every process undergoes a “dry run” phase where the algorithm mimics the loop nests of the QR algorithm down to the PBLAS level without actually applying modifications to or exchanging data. When the algorithm reaches the original point of failure, the matrix content is loaded from the checkpoint data and the algorithm is able to continue in the same manner as before in the FT-MPI based code. The regular recovery procedure is applied: the current iteration of the factorization is completed to update all checksums and the dataset is rebuilt using the ABFT reduction.

4.6 CoF Performance

In this section, we use our Open MPI and ABFT modifications to evaluate the performance of the CoF protocol. We use two test platforms. The first machine, “Dancer,” is a 16-node, development cluster in the Innovative Computing Laboratory at the University of Tennessee. All nodes are equipped with two 2.27GHz quad-core E5520 CPUs, with a 20GB/s Infiniband interconnect. Solid State Drive disks are used as the checkpoint storage media. The second system is the Kraken supercomputer, a University of Tennessee owned machine, housed at Oak Ridge National Laboratory. Kraken is a Cray XT5 machine, with 9,408 compute nodes. Each node has two Istanbul 2.6 GHz six-core AMD Opteron processors, 16GB of memory, and are connected through the SeaStar2+ interconnect. The scalable cluster file system “Lustre” is used to store checkpoints.

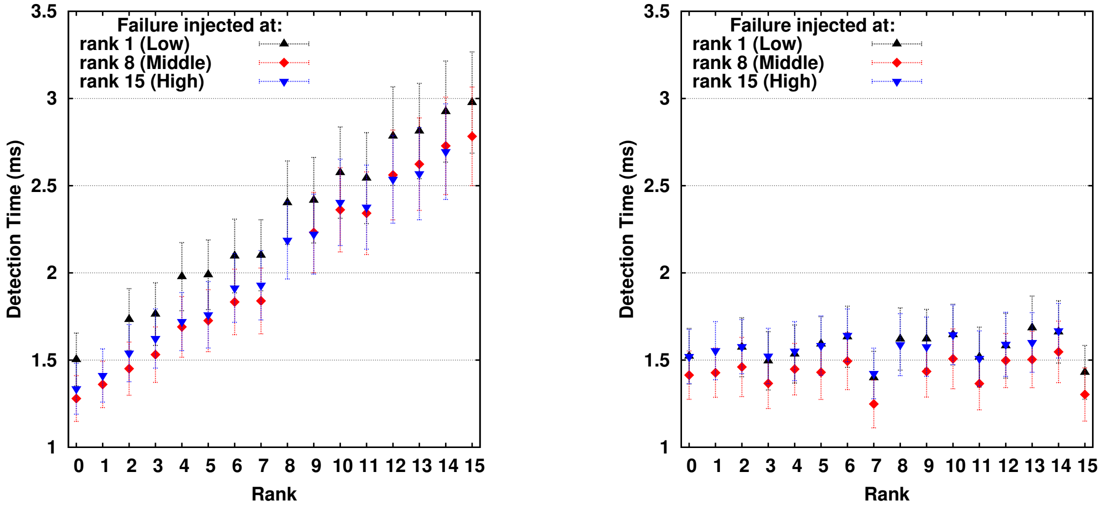
4.6.1 MPI Library Overhead

One of the main concerns from application developers when discussing fault tolerance is the amount of overhead introduced by the addition of fault tolerance into any application code or intermediate libraries. Our implementation of fault detection and notification is mostly implemented in the non-critical ORTE runtime. Typical HPC systems feature a separated service network (usually Ethernet based) and a performance interconnect; hence health monitoring traffic, which happens in the OOB service network, is physically separated from the MPI communications, removing the possibility of introducing network jitter due to fault tolerance messages. In addition, changes to the MPI functions are minimal: the same condition that previously triggered unconditional aborts has now been repurposed to trigger error handlers. As expected, no impact on MPI bandwidth or latency was measured. The memory usage of the MPI library is slightly increased, as the incarnation number doubles the size of the process names. However, this is negligible in typical deployments.

4.6.2 Failure Detection

Critical to the functionality of CoF is the reliable and expedient detection of process failures. The asynchronous failure notification described in Section 4.4.2, provides this failure detection. We designed a micro-benchmark to measure failure detection time as experienced by MPI processes. The benchmark code synchronizes with an `MPI_BARRIER`, stores the reference time, injects a failure at a specific rank, and enters a ring algorithm until the MPI error handler stores the detection time. The OOB routing topology used by the ORTE runtime introduces a non-uniform distance to the failed process, hence failure detection time experienced by a process may vary with the position of the failed process in the OOB topology.

Figures 4.2(a) and 4.2(b) present the failure detection times of the linear and binomial OOB topologies, respectively. The curves "Low, Middle, and High" show the behavior for failures happening at different positions in the OOB topology with



(a) Linear OOB Routing

(b) Binomial OOB Routing

Figure 4.2: Failure detection time, sorted by process rank, depending on the OOB overlay network used for failure propagation.

“Low” failures being injected at rank 1, “Middle” failures occurring at rank 8, and “High” failures at rank 15. On the horizontal axis is the rank of the detecting process, and on the vertical axis is the detection time experienced. The experiment uses 16 nodes, with one process per node, MPI over Infiniband, OOB over Ethernet, an average of 20 runs, and the MPI barrier latency is four orders of magnitude lower than the measured values.

In the linear topology (Figure 4.2(a)), every runtime process is connected to the *mpirun* process. For a higher rank, failure detection time increases linearly because it is notified by the *mpirun* process only after the notification has been sent to all lower ranks. Obviously, this OOB topology is not designed to be a scalable solution.

The binomial tree topology (Figure 4.2(b)) exhibits a similar best failure detection time. However, this more scalable topology has a low output degree and eliminates most contentions on outgoing messages, resulting in a more stable, lower average detection time, regardless of the failure position. Overall, failure detection time is on the order of milliseconds, a much smaller figure than the typical checkpoint time.

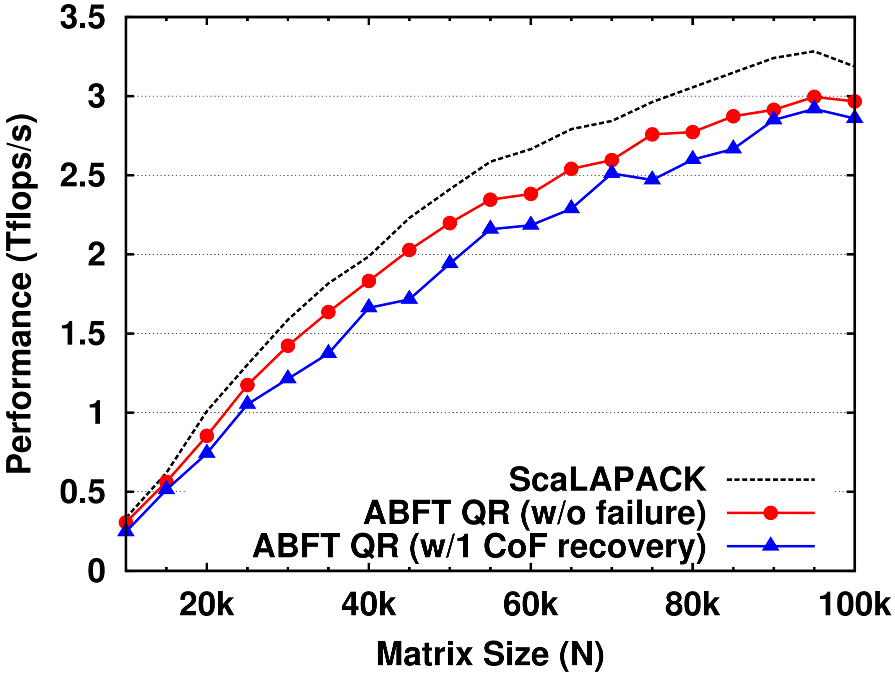


Figure 4.3: ABFT QR and one CoF recovery on Kraken (Lustre).

4.6.3 Checkpoint-on-Failure QR Performance

Supercomputer Performance: Figure 4.3 presents the performance on the Kraken supercomputer. The process grid is 24 × 24 and the block size is 100. The ABFT QR (w/o failure) curve presents the performance of the ABFT QR implementation, using CoF techniques, in a fault-free execution; it is noteworthy that when there are no failures, the performance is exactly identical to the performance of the unmodified ABFT QR implementation with FT-MPI. The ABFT QR (w/1 CoF recover) curve presents the performance when a failure is injected after the first step of the PDLARFB kernel. The performance of the non-fault tolerant ScaLAPACK QR is also presented for reference.

Without failures, the performance overhead compared to the regular ScaLAPACK is caused by the extra computation to maintain the checksums inherent to the ABFT algorithm [32]; this extra computation is unchanged between ABFT-QR without failures and ABFT-QR with a failure. Only on runs when a failure happened do

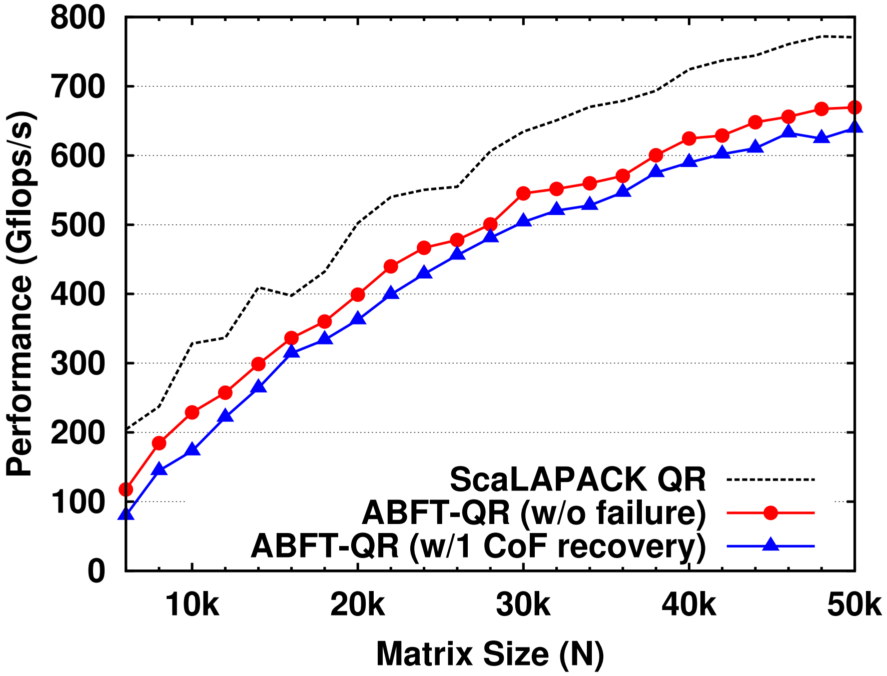


Figure 4.4: ABFT QR and one CoF recovery on Dancer (local SSD).

the CoF protocols undergo the supplementary overhead of storing and reloading checkpoints. However, the performance of the CoF-QR remains very close to the no-failure case. For instance, at matrix size N=100,000, CoF-QR still achieves 2.86 Tflop/s after recovering from a failure, which is 90% of the performance of the non-fault tolerant ScaLAPACK QR. This demonstrates that the CoF protocol enables efficient, practical recovery schemes on supercomputers.

Impact of Local Checkpoint Storage: Figure 4.4 presents the performance of the CoF-QR implementation on the Dancer cluster with an 8 × 16 process grid. Although a smaller test platform, the Dancer cluster features local storage on nodes and a variety of performance analysis tools unavailable on Kraken. As expected, the ABFT method has a higher relative cost on this smaller machine, but compared to the Kraken platform, the relative cost of CoF failure recovery is smaller on Dancer. Like all algorithms involving checkpointing, the CoF protocol incurs disk access overheads to store and load checkpoints when a failure hits, hence the recovery overhead depends

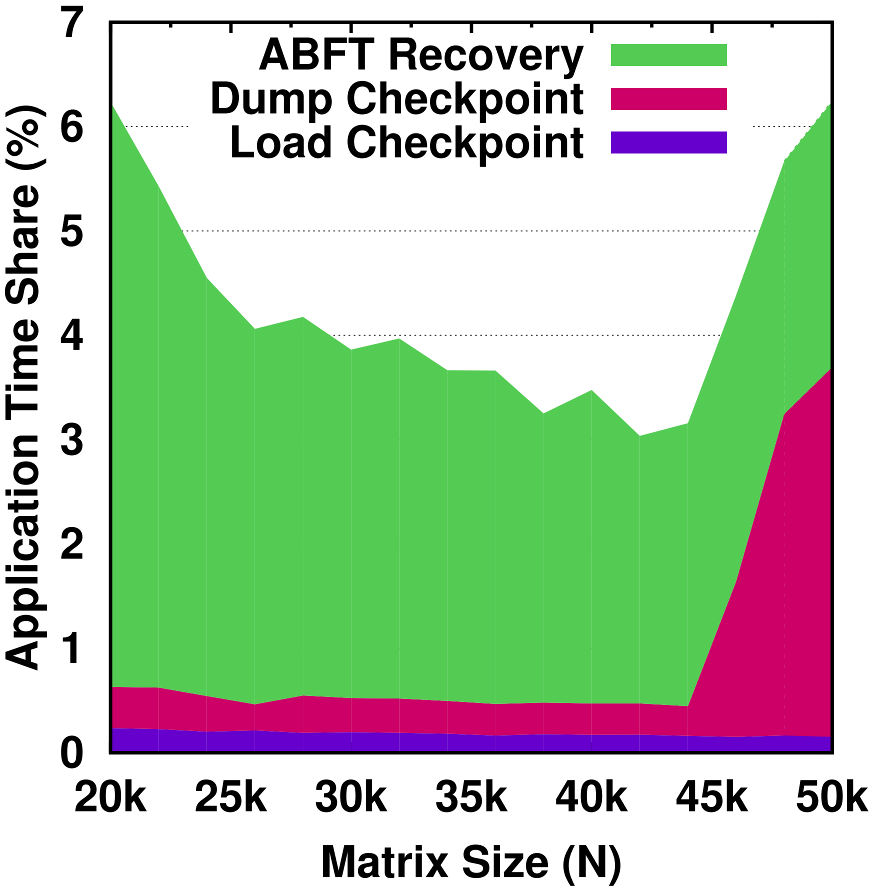


Figure 4.5: Time breakdown of one CoF recovery on Dancer (local SSD).

on I/O performance. By breaking down the relative cost of each recovery step in CoF, Figure 4.5 shows that checkpoint saving and loading only take a small percentage of the total run-time, thanks to the availability of solid state disks on every node. Since checkpoint reloading immediately follows checkpointing, the OS cache satisfies most disk access, resulting in high I/O performance. For matrices larger than N=44,000, the memory usage on each node is high and decreases the available space for disk cache, explaining the decline in I/O performance and the higher cost of checkpoint management. Overall, the presence of fast local storage can be leveraged by the CoF protocol to speedup recovery (unlike periodic checkpointing, which depends on remote storage by construction). Nonetheless, as demonstrated by the efficiency on

Kraken, while this is a valuable optimization, it is not a mandatory requirement for satisfactory performance.

4.7 Evaluation of CoF

Clearly, CoF does not meet all of the goals from Chapter 3. It provides relatively little **flexibility** due to the fact that it can only write a checkpoint after discovering a failure. It does not provide a platform to build other fault tolerance solutions. It provides sufficient **resilience** as it does allow the application to repair its execution by reloading the data it writes after a failure. This allows the application to continue executing even after a failure. Its greatest strength lies in its **productivity**. CoF is supported by existing MPI libraries and uses the familiar checkpointing paradigm as its basis. To that end, it is easily adopted by current applications as a possible solution for fault tolerance. A number of linear algebra algorithms can quickly adopt the tools in CoF with relatively little modification: one-sided factorizations, iterative conjugate gradient methods, and two-sided factorizations. The only changes necessary are to minimize the checkpoint size and to write a function to algorithmically repair the missing data.

Chapter 5

User Level Failure Mitigation

After creating CoF (see Chapter 4), it became clear that designing a fault tolerance framework within the existing MPI Standard to meet the goals in Chapter 3 would not be feasible. At that point, we investigated new ideas for fault tolerance that would require amendments to the MPI Standard, and we created a proposal for a new chapter called User Level Failure Mitigation. The complete document submitted to the MPI Forum is provided in Appendix A.

5.1 ULFM Design

Keeping with our design goals, we constructed a minimal set of new MPI constructs which would provide the necessary changes to the MPI Standard to allow applications to utilize fault tolerance in a way that makes sense to each one individually, rather than defining a uniform recovery mechanism. To provide additional capabilities and conveniences, we encourage the creation of libraries (see Chapter 6).

5.1.1 Failure Reporting

Failure reporting is essential for fault tolerance. Applications must be informed of failures from the MPI library in a consistent and predictable way in order to construct recovery mechanisms. The alternative would cause the application to be aware of some failures and oblivious to others, leading to a deadlock between processes. To that end, we decided to report failures using the return codes from existing MPI operations. This has the double benefit of being easy to understand from an application perspective and compliant with existing MPI constructs. Applications need only ensure that they check return codes for all MPI operations and act appropriately, an action which, ideally, they should already be taking, but in practice is not the current standard procedure.

The fundamental error code that applications will receive to be alerted to a process failure is `MPI_ERR_PROC_FAILED`. Another error code will be introduced later. When an application receives an error code related to a process failure, it indicates that the operation could not be completed successfully because an error occurred on one of the processes involved in the operation. This definition was specifically crafted to convey two ideas: 1) if an error causes a failure which prevents an MPI operation from completing for a process, that process must return an error code to report the failure; 2) if the operation *can* complete despite the failure for any reason (the communication involved in the operation is already finished, the implementation was able to circumvent the impact of the failures, etc.), it should do so and return no error code related to the process failure. Thus, knowledge of process failures is not global, but is local to any process which receives an error code indicating it.

Using local rather than global failure notification has substantial positive performance implications. Considering the alternative: if an error causes a failure, all MPI processes must report the same return code to the application to ensure global knowledge of the system. This forces each MPI operation to conclude with an agreement operation to determine the success or failure of the operation on all other processes. The current best known agreement algorithm has a runtime of $O(nlog(n))$ [42], which is a large amount of overhead to add to all MPI operations, some of which currently exhibit a runtime of $O(log(n))$. By only enforcing global

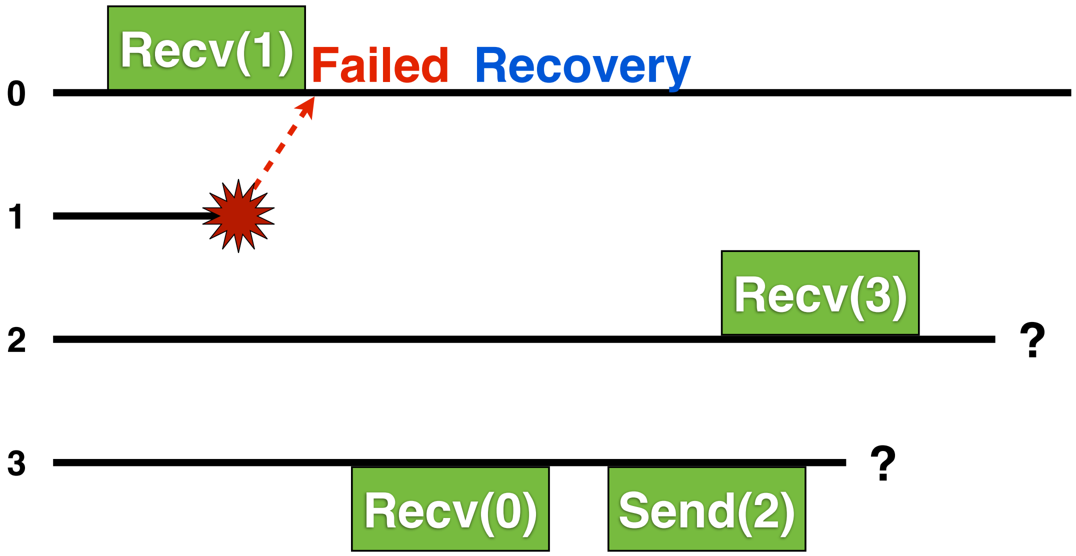


Figure 5.1: Application discovers failure and encounters deadlock

knowledge when the user requires it, we bypass this increased overhead and keep the per-operation cost low.

Though it is now established that global knowledge of failure is expensive and therefore should not be imposed on all applications, there are times where such knowledge is necessary. The most obvious example of such a time is during recovery. Figure 5.1 demonstrates an example of a recovery situation where local knowledge is not sufficient to prevent deadlock. In this example, four processes are communicating sporadically when process 1 fails. Process 0 immediately discovers the failure because it is actively communicating with process 1 at the time. Process 0 branches from the normal execution and begins recovery operations. In the meantime, processes 2 and 3 are dependent on communications from process 0 in order to continue their own execution. They enter blocking operations where they become deadlocked. Because global knowledge of failures did not exist, the processes did not know to enter recovery operations or to cancel their outstanding communication operations and transition to recovery.

To resolve this situation, we introduce the new MPI construct: `MPI_COMM_REVOKE`. This operation is a non-local, non-collective operation to propagate failure information throughout an MPI Communicator. It does this by using an out-of-band, resilient broadcast algorithm to interrupt all other non-local MPI operations and return the new error code `MPI_ERR_REVOKED`. In this sense, it works similarly to the existing MPI function, `MPI_ABORT`, without the subsequent ending of the MPI application. Both `MPI_ERR_REVOKED` and the previously introduced error code, `MPI_ERR_PROC_FAILED`, are permanent errors in the sense that once one of these codes are returned, the MPI Communicator will never be usable again for interprocess communication, though it is possible to transition from the error code `MPI_ERR_PROC_FAILED` to `MPI_ERR_REVOKED` after the function `MPI_COMM_REVOKE` is called.

It is important to understand that `MPI_COMM_REVOKE` has no matching call on remote processes. Once a process calls it on a particular MPI Communicator, all other processes in the communicator will eventually receive the notification of revocation through the error codes of other MPI operations as if the function was called on their local processes. If another MPI process never makes another call to an MPI operation, it will never be notified of the revocation of the communicator. If it is necessary for all processes to be aware of process failures in this scenario, we provide a tool in Section 5.1.5 to build such stronger consistency.

By reexamining the scenario introduced earlier, now in Figure 5.2, we can see how this function might be used. After the failure of process 1, process 0 invokes `MPI_COMM_REVOKE`. While processes 2 and 3 have already entered their respective communication operations, the notification that their communicator has been revoked causes those `MPI_RECV` operations to return with the error code `MPI_ERR_REVOKED`. At this point, all processes can perform recovery together and the deadlock scenario in Figure 5.1 is averted.

Though `MPI_COMM_REVOKE` can appear to be a useful catchall tool to introduce global knowledge of failures to all applications, a better understanding of the tool emphasizes that it should be used carefully and only in scenarios where it is required.

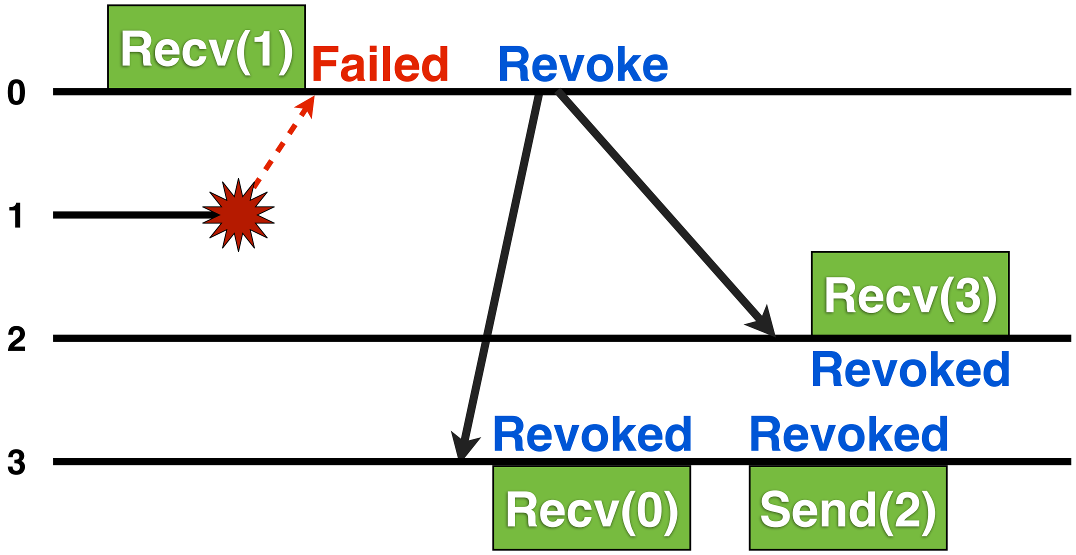


Figure 5.2: Application discovers failure and recovers using `MPI_COMM_REVOKE`

Not all applications require global knowledge of failures, and introducing it manually can impose a large synchronization and recovery overhead that would be otherwise unnecessary. An example of such a scenario is a Monte Carlo, master-worker style application. The usual communication pattern of such applications is that all worker processes communicate with a master process but not with each other. Thus, there are many point-to-point communications, but no collective communications. These applications should not use `MPI_COMM_REVOKE` to alert the worker processes to failure of another worker, but should simply continue their point-to-point communications unchanged. This example shows the flexibility of ULFM by not imposing an automatic recovery mechanism. In some cases, no recovery is the best course of action.

5.1.2 Rebuilding Communicators

When collective communication is required, using an existing MPI communicator with failed processes is no longer an option. In this case, a new MPI construct to restore the ability to communicate is necessary. To facilitate this, we created the new function, MPI_COMM_SHRINK. This function is similar to the automatic repair method of the same name used by FT-MPI [34]. After an MPI communicator has been revoked, the remaining alive processes must call this function collectively. The shrink operation will create a new MPI communicator by executing an agreement algorithm among all alive processes to determine the group of processes which are believed to have failed. This failed group is excluded and a new MPI communicator is created with the remaining processes. The new communicator does not replace the revoked communicator, but is provided as a new MPI communicator with a new handle. This facilitates easier recovery by allowing the application to reference the previous "version" of the communicator to acquire information such as previous size, rank, and various other communicator properties.

Of all of the new MPI constructs, MPI_COMM_SHRINK was the most difficult to design. Many options for communicator repair and recovery were considered before deciding on shrink, some inspired by the recovery modes in FT-MPI. We will mention some of the alternatives here to demonstrate the rationale behind the design decisions.

Blank

The most seriously considered alternative to MPI_COMM_SHRINK was an operation to replace failed processes with MPI_PROC_NULL, introducing blank positions within the MPI communicator. The advantage of this scenario is that all processes retain their existing ranks and topologies, making continuing execution after failure an easy transition as applications can continue to communicate in the same patterns as before the process failure. However, as with many of the communicator repair options, this mechanism does not provide the flexibility seen in shrink. For example, by removing processes from the communicator, they can no longer be queried to determine information about the communicator, such as the ranks of the failed processes. This would also introduce a complexity when trying to replace the failed processes. These complexities will be further examined below.

Replace

Replace was the default recovery mechanism in FT-MPI. Failed processes were automatically replaced with a new process in the same rank and location in the MPI communicator. For applications, this is the simplest form of recovery to understand because it automatically reconstructs MPI_COMM_WORLD and re-spawns failed processes. No manual recovery is required. However, from an implementation perspective, implementing this automatic recovery mechanism is very expensive and introduces many difficult problems related to communicator reconstruction. FT-MPI solved the problem of communicator reconstruction by destroying all communicators other than MPI_COMM_WORLD and requiring the application to reconstruct the communicators manually. This is a heavy-handed approach and is improved by using shrink. Also, as with the blank functionality, using the replace mechanism does not facilitate as many other forms of fault tolerance, but requires that applications conform to the decisions mandated by replace.

Using existing MPI functions

The last consideration was to modify the existing MPI functions to include fault-tolerant semantics. An example of a function where this would make sense is MPI_COMM_DUP. Semantically, the newly defined function would be similar to MPI_COMM_SHRINK, however redefining existing MPI constructs introduces both confusion and incompatibility. Existing MPI codes would be forced to be rewritten to either specifically handle process failures as defined by the new text or specifically exclude them, rather than the behavior we chose where applications are clearly either using fault-tolerant constructs, or are not, depending on whether or not they call the new recovery functions. Without very careful design, using existing functions as fault-tolerant mechanisms could cause confusion if failures occur in an inconvenient place. If a failure occurs just before MPI_COMM_DUP and the operation creates the new communicator without the failed process, an application which does not expect

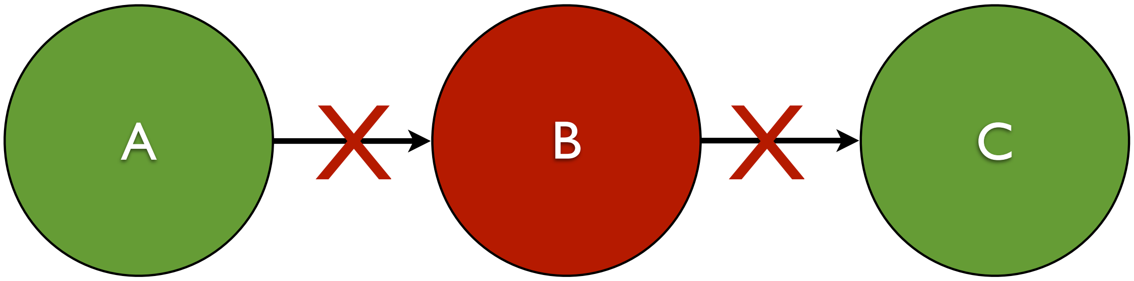


Figure 5.3: Intermediate node failures report as `MPI_ERR_PROC_FAILED`

the recovery would need to perform extra checks to see if the communicator has the expected size and composition. The same can be said for other MPI functions, such as the new function `MPI_COMM_CREATE_GROUP`, introduced by MPI-3.

5.1.3 Failure Discovery

Once a failure has been reported to the MPI processes and the processes have taken steps to disseminate knowledge as necessary, another issue must be addressed. The living processes will need to have a mechanism to discover which group of processes have actually failed and should be excluded from the continuing application. While it might seem obvious that the failed process would be the one with which the failed communication was taking place, this is not guaranteed to be the case. As an example (see Figure 5.3), if process A is communicating with process C and the communication topology routes messages through the node containing process B, a failure of process B could result in an inability to communicate between processes A and C. While a good MPI implementation should make every effort to solve such routing issues transparently, it is possible that a scenario would occur where such bifurcation is unavoidable or the implementation chooses not to repair the communication paths. In this case, a point to point communication operation between A and C would return the error code MPI_ERR_PROC_FAILED, even though neither A nor C has actually failed. To discover the actual source of the failure, a new set of functions is necessary.

The functions provided for this purpose are MPI_COMM_FAILURE_ACK and MPI_COMM_FAILURE_GET_ACKED. By calling this set of functions, the application can acquire the MPI Group containing the set of processes which are known to have failed. MPI_COMM_FAILURE_ACK sets a reference point within the MPI implementation to which MPI_COMM_FAILURE_GET_ACKED refers back when determining the group of failed processes. This group represents only local knowledge and is not guaranteed to be uniform among all process. No matter how many times the MPI_COMM_FAILURE_GET_ACKED is called, the group of failed processes will not change until the reference point is changed by calling MPI_COMM_FAILURE_ACK. By splitting the functions in two this way, the application can maintain thread safety by controlling failure knowledge between the threads.

5.1.4 Wildcard MPI Receive Operations

The other benefit of splitting the operation of acquiring the group of failed processes into two functions is that the MPI_COMM_FAILURE_ACK function has another purpose. MPI contains a constant, MPI_ANY_SOURCE, which can be used to specify that a receive operation should match a message coming from any other rank within a communicator rather than the usual format where a specific source is provided. When considering failure scenarios and knowledge of the status of the ranks, this presents a difficult situation for the application. If a failure needs to be reported during such a wildcard receive operation, MPI_ERR_PROC_FAILED is not an accurate representation of the status of the operation. While a process involved in the operation will have failed, it might not be the one with which the wildcard receive would have matched. In this case, it is still important that we alert the application to a possible failure, but we should also provide a way for the application to continue to use the wildcard receive constant after the failure notification so as to not require an expensive, complete

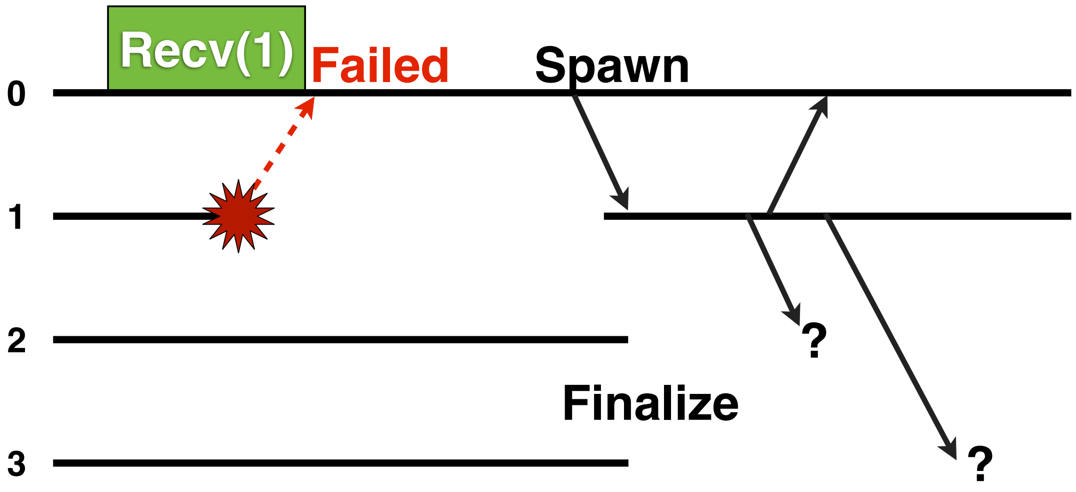


Figure 5.4: Application reaches inconsistent state after some processes exit before other processes

recovery. In this situation, we return the error code MPI_ERR_PENDING to inform the application that the receive operation is still pending and can be completed after the application acknowledges the failure using MPI_COMM_FAILURE_ACK. Once the application acknowledges the failure, the MPI library will not return another error code related to that specific process failure and the application can re-enter the wildcard receive operation. It should be noted that MPI_ERR_PENDING is not a new error code, but the existing definition, “pending request”, applies to this scenario and so defining a new error code was decided to be unnecessary.

5.1.5 Process Consistency

While a large focus of the ULFM work has been to provide a system with weak consistency between processes to improve performance, there are times where stronger consistency is necessary. Figure 5.4 demonstrates a situation where this could occur. Processes 2 and 3 believe the application is completed and call MPI_FINALIZE while process 1 fails, to be discovered by process 0. When process 0 spawns a replacement for process 1 and the two processes try to perform recovery operations, the data needed from processes 2 and 3 is no longer available. In this situation, an agreement algorithm among the processes is necessary before algorithm completion to ensure that all processes successfully reach MPI_FINALIZE. While in this scenario, using an existing MPI function such as an MPI_ALLREDUCE could solve the problem, if the application does not need to recover process 1, the collective operation would no longer complete successfully without repairing the communicator, an expensive and possibly unnecessary operation.

To provide a tool to resolve this scenario, we created MPI_COMM_AGREE. This function performs a fault tolerant agreement algorithm over a boolean value among all alive processes. All alive processes participate with the value passed in as an argument and all dead processes do not participate (which is semantically equivalent to participating with the value *TRUE*). This allows applications which communicate only via point to point operations to complete the agreement algorithm despite process failures. The rationale behind ignoring process failures (including new process failures) is that if the failure had impacted an MPI communication, that operation would have returned an error code reporting the failure. If none of the processes detected that failure, then it did not impact the results and can therefore be safely ignored. If an error code was previously reported due to process failure, the process which received it can participate with the value *FALSE* and all other processes will know that they need to enter recovery operations.

The only error code related to process failure that MPI_COMM_AGREE may return is MPI_ERR_REVOKED. If the communicator has been revoked remotely (or indeed locally), it is most likely that the revoke operation was intended to interrupt even the finalization operation and that recovery is necessary.

5.2 Beyond Communicators

While communicator operations are the most common use of MPI and historically the core of MPI, there are other chapters of the MPI Standard that cover other communication contexts. Dynamic processes, shared memory windows and collective file I/O operations have been included in the Standard to provide increased functionality in the form of one-sided communication operations and large scale file manipulation. The work to provide added fault tolerance capabilities to these types of operations will continue as this work was removed from the original proposal to the MPI Forum, but the fundamentals detailed here are similar to the fault tolerance designed for MPI Communicators.

5.2.1 Failure Notification

Well-defined failure notification is key to managing failures. By defining the expected behavior of the MPI library after a failure, the application can design resilience protocols to ensure a deadlock-free execution.

Dynamic Processing

Dynamic processing requires that the MPI implementation construct new communicators, called intercommunicators, to connect the group of existing processes to the group of new processes. The most critical requirement is that if a process failure is detected while the MPI library is constructing the new MPI intercommunicator, the MPI library must always notify the root processes on either side of the intercommunicator. This is key because collective and point-to-point communication from the local group of an intercommunicator to a remote group is expensive at best and impossible at worst after a failure has impacted the capabilities of the communicator. By ensuring the root processes have been notified, they can provide notification to all processes in their local groups by revoking their communicator. Also, for similar reasons, when creating new processes by calling `MPI_COMM_SPAWN`, if a process failure is detected during the process or communicator creation, the MPI library should not return a partially constructed MPI communicator which cannot be repaired. Instead, the library should return a communicator that does not function (such as MPI_COMM_NULL) which will signify to the new MPI processes that they should probably abort, and to the old processes that they should attempt to spawn new processes again.

One-Sided Communication

One-sided communication operations behave as non-blocking communication operations. Because of this, they have similar failure notification definitions to calls involving MPI communicators such as MPI_ISEND and MPI_IRECV. Rather than notifying applications of process failure during the initialization calls, the library should delay notification to the one-sided completion calls (i.e. MPI_WIN_COMPLETE, MPI_WIN_WAIT, etc.). Remote memory access calls also have different failure semantics than traditional MPI communicator-based operations, though the motivation is similar. Whenever a failure is reported during one-sided epochs, the memory targeted during that period is undefined. This definition is similar to the fact that any buffers used during MPI operations where a failure is reported are also undefined. More specifics of the failure notification definitions for remote memory operations can be found in the specification document in Appendix A.

File I/O

File I/O operations are more complex when considering failure notification. Unlike communicator-based operations and one-sided communication operations, they usually do not involve synchronization calls which would facilitate failure propagation and notification. Because of this, after a failure the state of any open file pointers is left undefined. To help mitigate this indefinite result, users are encouraged to use a communicator which contains the same group of processes as those involved in an

MPI file object to ensure that all processes remain functional after critical I/O calls. While this does create more overhead for file operations, it is less overhead than would be introduced by redefining the I/O operations to synchronize on each operation to ensure that all processes were successful.

5.2.2 ULFM Functions for One-Sided Communication

To assist with failure notification and recovery, two new functions have been introduced for MPI one-sided communication. These functions closely mirror similar functions found in the communicator-based recovery Section 5.1.

The first function, `MPI_WIN_REVOKE` is identical to `MPI_COMM_REVOKE`, but with MPI windows rather than MPI communicators. When any process involved in a window calls `MPI_WIN_REVOKE`, all other processes are eventually notified by receiving the error code, `MPI_ERR_REVOKED`. From this point on, all non-local operations must continue to return the error code `MPI_ERR_REVOKED`. As with `MPI_COMM_REVOKE`, this operation is both non-local and non-collective, meaning that if any process in the window calls the function, it impacts all other processes without a matching call.

The second function, `MPI_WIN_GET_FAILED` provides a mechanism to retrieve the group of processes which are locally known to have failed at the time of calling. Note that this function is similar to the combined meaning of `MPI_COMM_FAILURE_ACK` and `MPI_COMM_FAILURE_GET_ACKED`. The reason these functions are combined is that there is no need to acknowledge the failure to allow wildcard operations to complete. One-sided communication does not contain a similar operation.

5.2.3 ULFM Functions for File I/O

As with failure notification, recovery functions for File I/O are also minimal. The only function provided is `MPI_FILE_REVOKE`. Again, this function is non-local and non-collective and will notify all other processes involved in the MPI file and cause all subsequent non-local operations to return the error code `MPI_ERR_REVOKED`.

When deciding whether to include a function to retrieve the failed processes from the file object, it was decided that a valid reference point to describe the failed processes does not exist. In communicators and windows, it is possible to retrieve the group of processes involved in the communication object. However, for file objects, this option is not available. This makes describing the failed processes difficult and the need for a function such as `MPI_FILE_GET_FAILED` unnecessary.

5.3 ULFM in Applications

ULFM has already been ported to some existing applications to demonstrate its usability in real-world scenarios. This section describes its use in the linear algebra ABFT-QR algorithm detailed in Section 4.5.

5.3.1 Example: QR-Factorization

This example is another proof of concept demonstration as in Section 4.5. The ABFT algorithm for QR factorization is the same, but the recovery technique changes to incorporate the new capabilities of ULFM. Rather than dumping the checkpoints to disk, restarting the MPI job, re-reading the data from disk, and continuing the execution, the ULFM version of ABFT-QR does not need to perform all the tasks to restart MPI. Instead, remaining processes can start a replacement for the failed process and perform forward recovery without needing to restart the existing MPI processes.

When a process failure is detected in the QR algorithm, an error is returned to the application. The application revokes the communicator being used by the BLACS (Basic Linear Algebra Communication Subroutines) [30] library to ensure that all processes are notified of the process failure (see the section on BLACS below for a description of the modifications to BLACS to enable fault tolerance). After revoking the communicator with the failed process, the remaining processes create a working communicator using MPI_COMM_SHRINK. Using the new communicator, they spawn a new process to replace the failed process. By using MPI_INTERCOMM_MERGE and MPI_COMM_SPLIT, they can reposition the new process to have the same rank as the original process which it replaced. This means that the ABFT algorithm can continue as before once the data recovery is complete.

BLACS

BLACS is the intermediate library used by ScaLAPACK to abstract the communication in the linear algebra codes. While at one time it provided an abstraction for many communication libraries and platforms (MPI, PVM, HP Exemplar, IBM SP (MPI), Intel series (NX), and SGI Origin 2000 (MPI)), as most of these libraries have become unnecessary and merged together, BLACS has reduced its set of supported libraries to only MPI. This simplified the changes needed to repair the library in the event of failure. BLACS already provides a function to use a custom MPI communicator as a basis for communication. By using this functions, the application can provide a working communicator at the beginning of execution and a repaired communicator after failures are detected and corrected.

The necessary modifications to BLACS were related to the fact that BLACS assumes MPI_COMM_WORLD to be a working and complete communicator. At the time of writing for BLACS, neither dynamic processes nor fault tolerance were being considered in the context of MPI and thus both assumptions are correct. Now that new processes can be spawned which no longer are in the scope of the original MPI_COMM_WORLD, and processes can fail which causes communicators to become unusable, these original assumptions are no longer valid.

To repair the internal information needed by BLACS, the user needs to call blacs_set to set the values of the processor's rank in the currently used communicator and the size of the communicator. After this, the remaining internal data used by BLACS can either be ignored or is repaired by providing a new communicator.

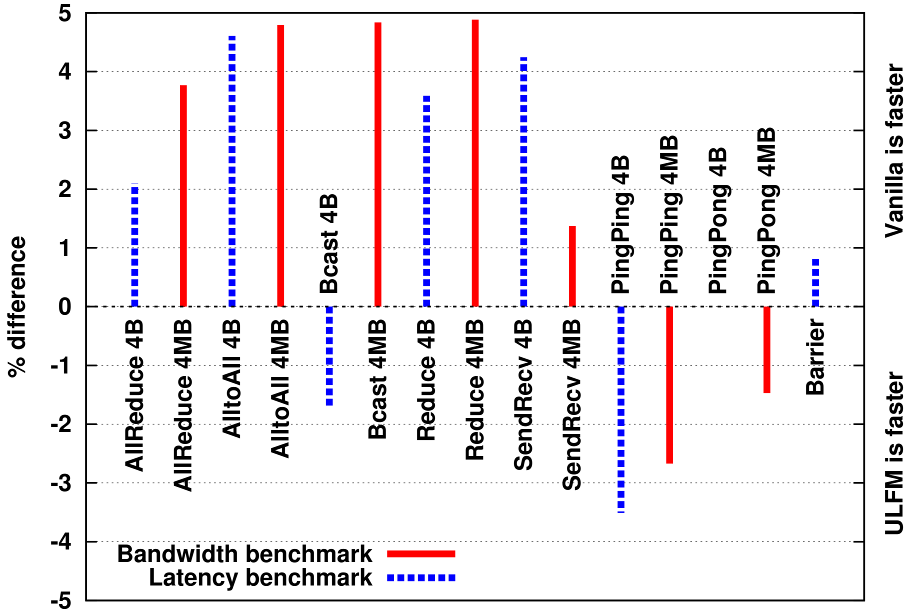


Figure 5.5: Relative difference between ULFM and Vanilla Open MPI on Shared Memory

5.4 ULFM Performance

Though the ULFM implementation is designed to be a reference implementation and therefore will not achieve the performance of a fully tested and supported MPI implementation, we do want to demonstrate that even under such conditions, reasonable performance can be expected. To evaluate the performance of ULFM, we have two main types of tests. The first set of tests will demonstrate that ULFM does not introduce a significant overhead to a failure-free execution by demonstrating latency and bandwidth tests. The second type of test will show the performance of ULFM when executing the ABFT-QR Factorization code described in Section 5.3.1.

5.4.1 MPI Overhead

The first set of tests demonstrates the overhead of the changes made in the MPI implementation. This work builds on the work demonstrated in Chapter 4 to support

Table 5.1: NetPIPE results on Smoky.

1-byte Latency (microseconds) (cache hot)

Interconnect	Vanilla	Std. Dev.	Enabled	Std. Dev.	Difference
Shared Memory	0.8008	0.0093	0.8016	0.0161	0.0008
TCP	10.2564	0.0946	10.2776	0.1065	0.0212
OpenIB	4.9637	0.0018	4.9650	0.0022	0.0013

Bandwidth (Mbps) (cache hot)

Interconnect	Vanilla	Std. Dev.	Enabled	Std. Dev.	Difference
Shared Memory	10,625.92	23.46	10,602.68	30.73	-23.24
TCP	6,311.38	14.42	6,302.75	10.72	-8.63
OpenIB	9,688.85	3.29	9,689.13	3.77	0.28

CoF. The runtime found in CoF is the same for both versions of MPI, so the performance discussions in Section 4.6.1 are also valid for ULFM.

Intel MPI Benchmarks

We start with a demonstration of latency and bandwidth using the Intel MPI Benchmark test suite [2]. This suite has many tests to measure the performance of everything from collective operations to latency times. We run this test using "Romulus", a large shared memory machine at the University of Tennessee. In Figure 5.5 we see that the impact of the ULFM changes to MPI are negligible, as expected. For tests where the default Open MPI performed better, the bar is above the center line, and for tests where ULFM had better performance, the bar is below the center line. For all of the tests, the relative difference remains below 5%, which is within the standard deviation of the tests on that machine, showing that any difference in the performance of the two implementations is negligible.

NetPIPE

The next test found in Table 5.1 uses the NetPIPE [6] benchmark (version 3.7) to measure the 1-byte latency and bandwidth of both Vanilla Open MPI and ULFM. Here again, we find that any difference between the two implementations is within both the noise limit of the network and the standard deviation of the test, showing

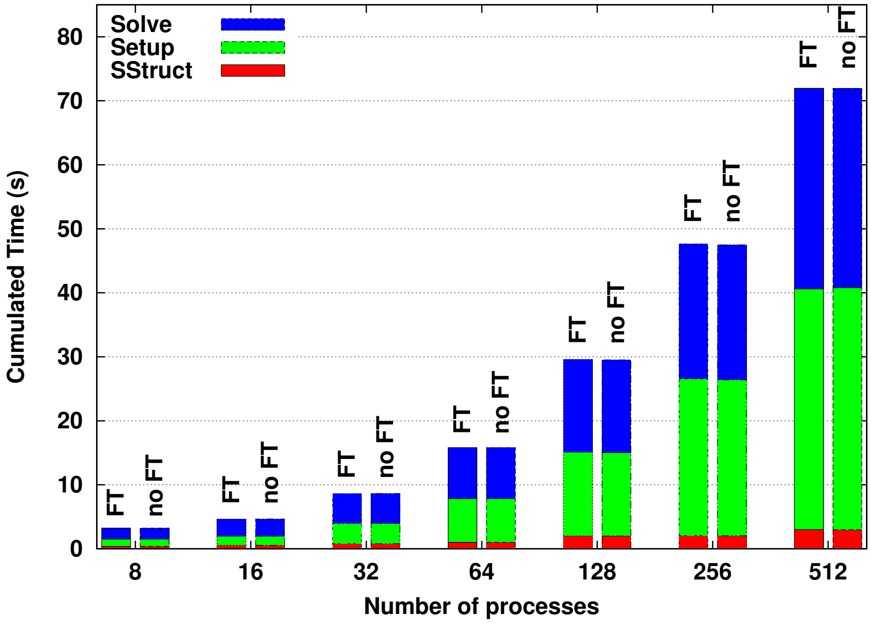


Figure 5.6: Comparison of Sequoia-AMG running at different scales with ULFM and Vanilla Open MPI

that the impact of the ULFM modifications on a failure-free MPI environment is minimal.

Sequoia-AMG

To demonstrate the impact of the ULFM modifications on a full application, we used the Sequoia-AMG [9] benchmark on "Smoky", a 512 node cluster at Oak Ridge National Laboratory where each node contains four quad-core 2.0 GHz AMD Opteron processors with 2 GB of memory per core. The benchmark is an Algebraic Multi-Grid (AMG) linear system solver for unstructured mesh physics and makes heavy use of MPI. We measured the weak scaling results and found that there was virtually no difference between the ULFM MPI performance and the Vanilla Open MPI performance. It is important to remember the difference between the type of results we see here and the results seen in Section 5.4.2. These tests do not contain modifications to undergo failure or process recovery, but are only measuring the overhead of the MPI implementation.

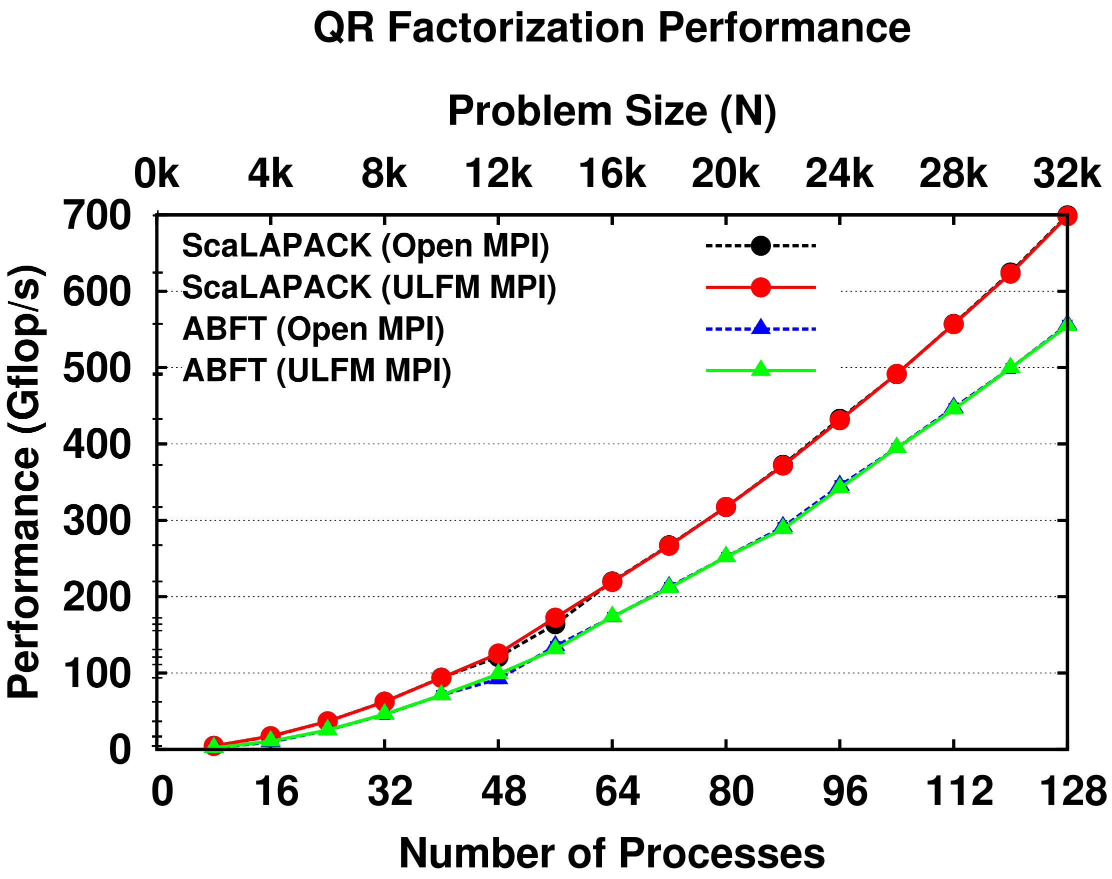


Figure 5.7: Weak-Scaling performance of ABFT-QR on Grid5000 'Graphene' compared to ScaLAPACK in both Vanilla Open MPI and the ULFM version

5.4.2 ABFT-QR Factorization

As discussed in Section 5.3.1, the ABFT-QR Factorization code has been modified to work with ULFM to demonstrate how ULFM can be leveraged to provide fault tolerance to a "real world" algorithm. Here we discuss the performance of the algorithm using a machine found in Grid'5000 [*]. We used the "Graphene" cluster at the Nancy site. "Graphene" is a 144 node cluster using Intel Xeon X3440 2.53 Ghz 4 core processors, 16 GB of memory, and Infiniband-20G cards. As with the IMB tests above, for all tests performed here, we used the *tcp* BTL.

[*] Acknowledgment: Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

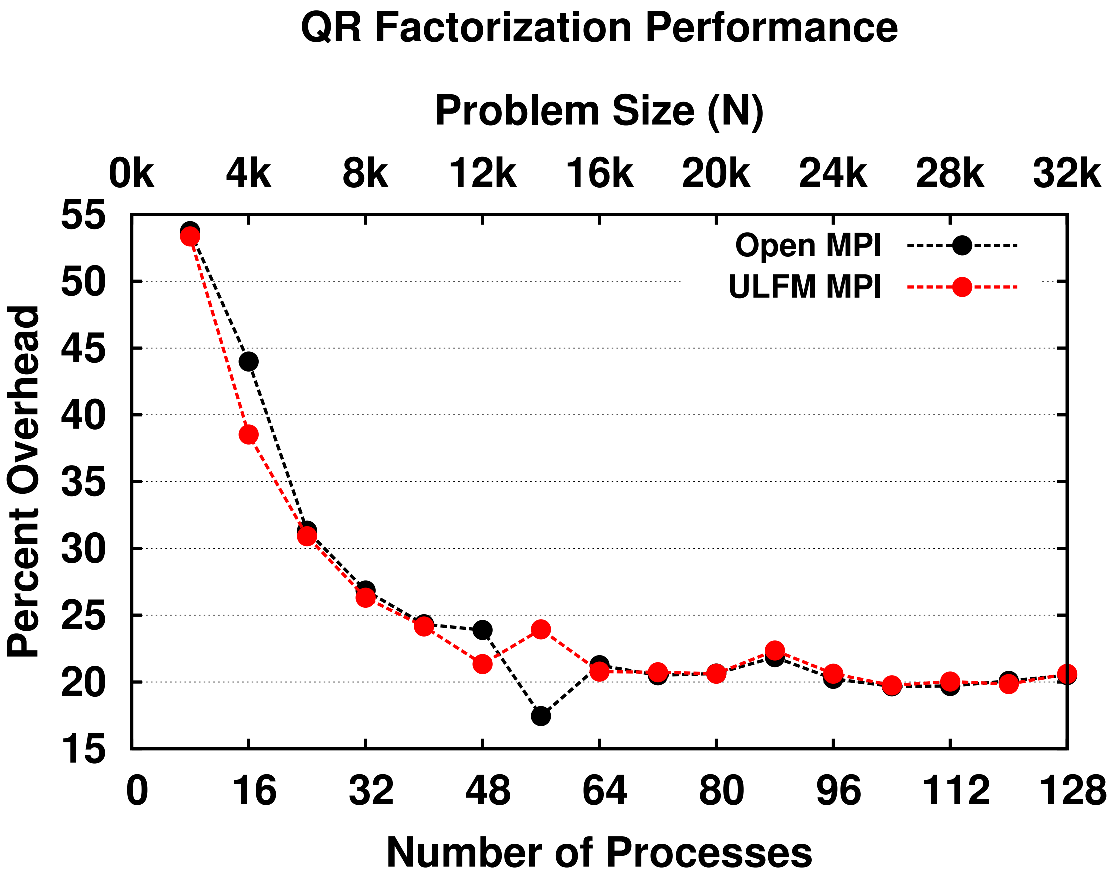


Figure 5.8: Overhead of ABFT with Vanilla Open MPI and ULFM MPI

The first test in Figure 5.7 demonstrates the performance overhead of our modifications to the ULFM library. In this graph, we see that our changes have almost no impact on the results of the test which we will use as an established fact for the remainder of the discussion. This test also demonstrates the weak scaling capability of the ABFT algorithm using both the original Open MPI library and the ULFM MPI library.

In Figure 5.8, we show the overhead of the ABFT algorithm itself using the same original data. While the gap appears to grow in Figure 5.7, Figure 5.8 shows that the relative difference between the two implementations actually stabilizes. This is the expected result for this type of test. For small problem sizes, the overhead will be high because the problem is not large enough to fill the pipeline of the machines.

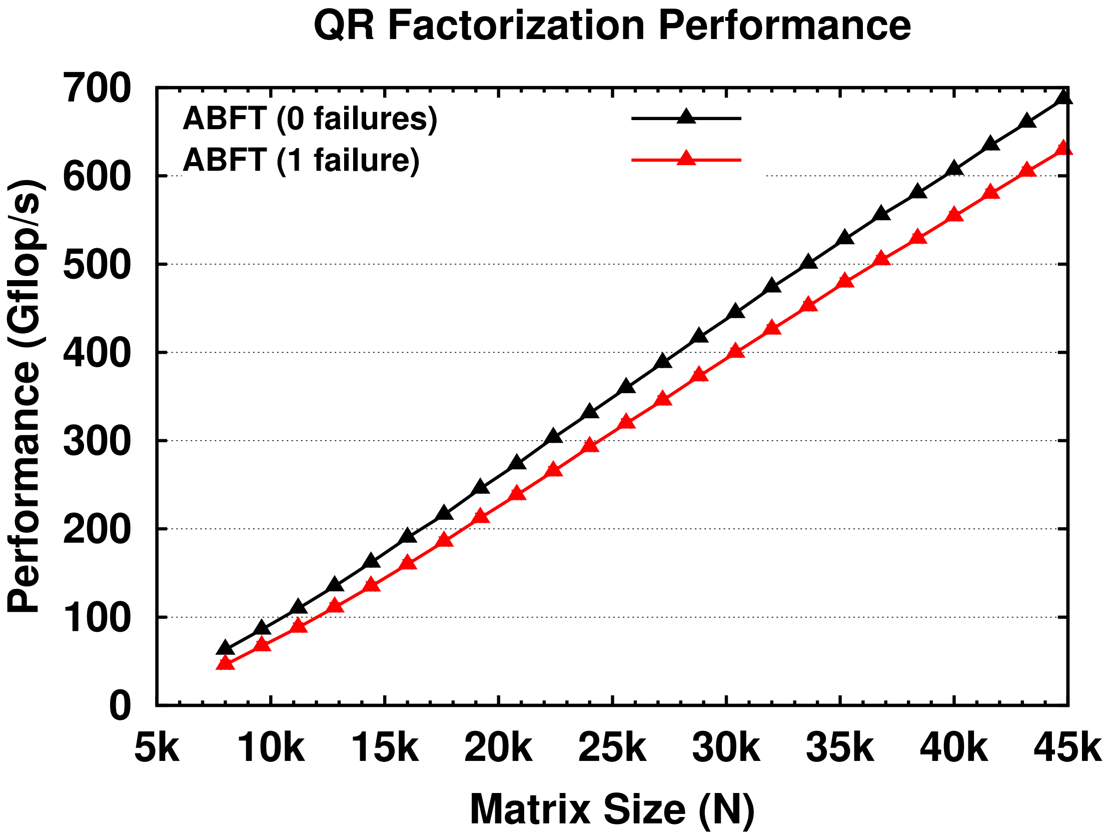


Figure 5.9: Strong-Scaling performance of ABFT-QR on Grid5000 'Graphene' with no failures and one failure

However, as the problem size increases, the experienced overhead stabilizes at around 20% of the execution time. While we expect that through more optimization, this overhead could shrink, perhaps significantly, it will never disappear entirely as the ABFT algorithm will always incur overhead from the data protection schemes.

Now that the overhead of the MPI implementation itself and the ABFT code has been established, the most interesting result of the QR factorization test is to demonstrate the overhead of the failures themselves. To show this, we used a strong scaling test, where the number of processes is held steady at 128 nodes and the problem size increases from 8,000 to 44,000. The results of this test are not directly comparable to the results of the previous tests as the configuration of the MPI library was slightly altered. Figure 5.9 demonstrates the performance of the ABFT algorithm

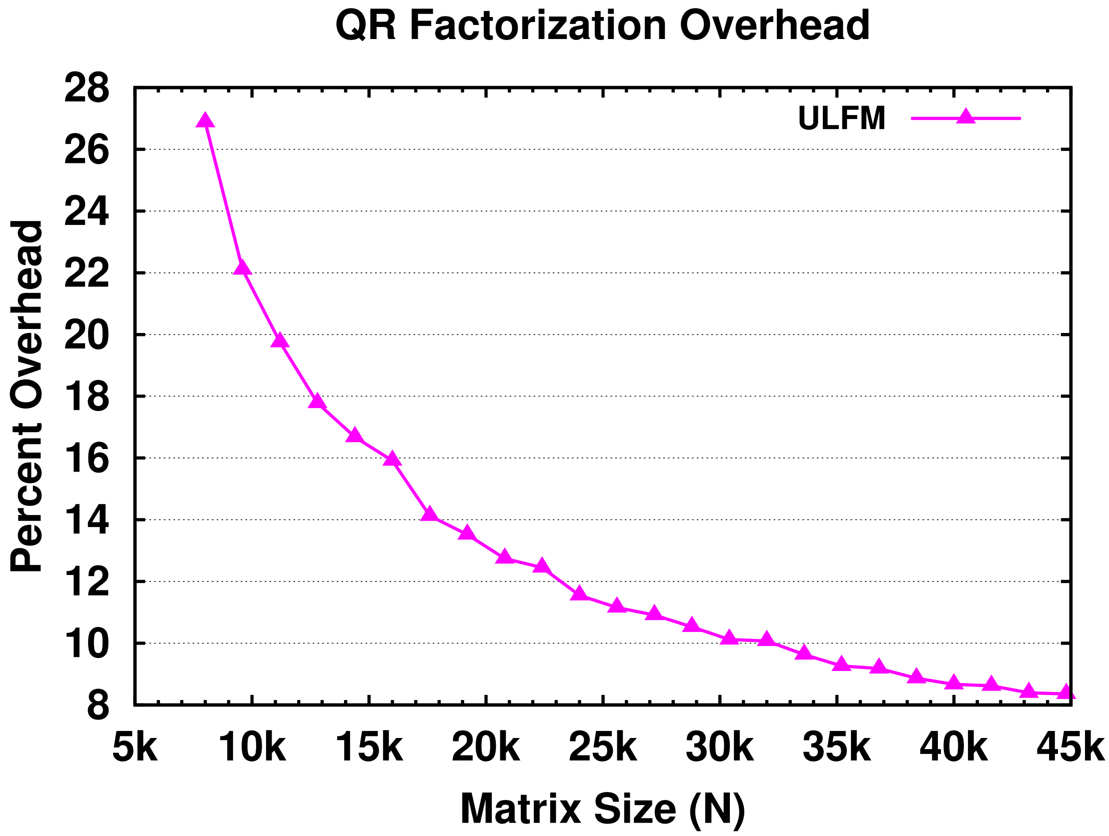


Figure 5.10: Overhead of one failure with ABFT-QR on ULFM MPI

with no failures and again with one failure using our ULFM implementation of MPI. Here we see that the relative overhead of the failure seems to be relatively low and the algorithm still achieves good performance.

We quantify this overhead in Figure 5.10 where we see that the overhead of the failure diminishes quickly to close to 8%, though it would probably continue to drop at larger scales. We expect this value to decrease due to the relatively low overhead of failure recovery. The cost of recovery itself (as opposed to the data protection built into the ABFT algorithm) is only the cost of replacing the failed process and repairing the data in the matrix. These costs are detailed in Figure 5.11. The MPI recovery time stays constant as the matrix size increases and the number of processes remains constant. This number stays relatively low at around 400 milliseconds to

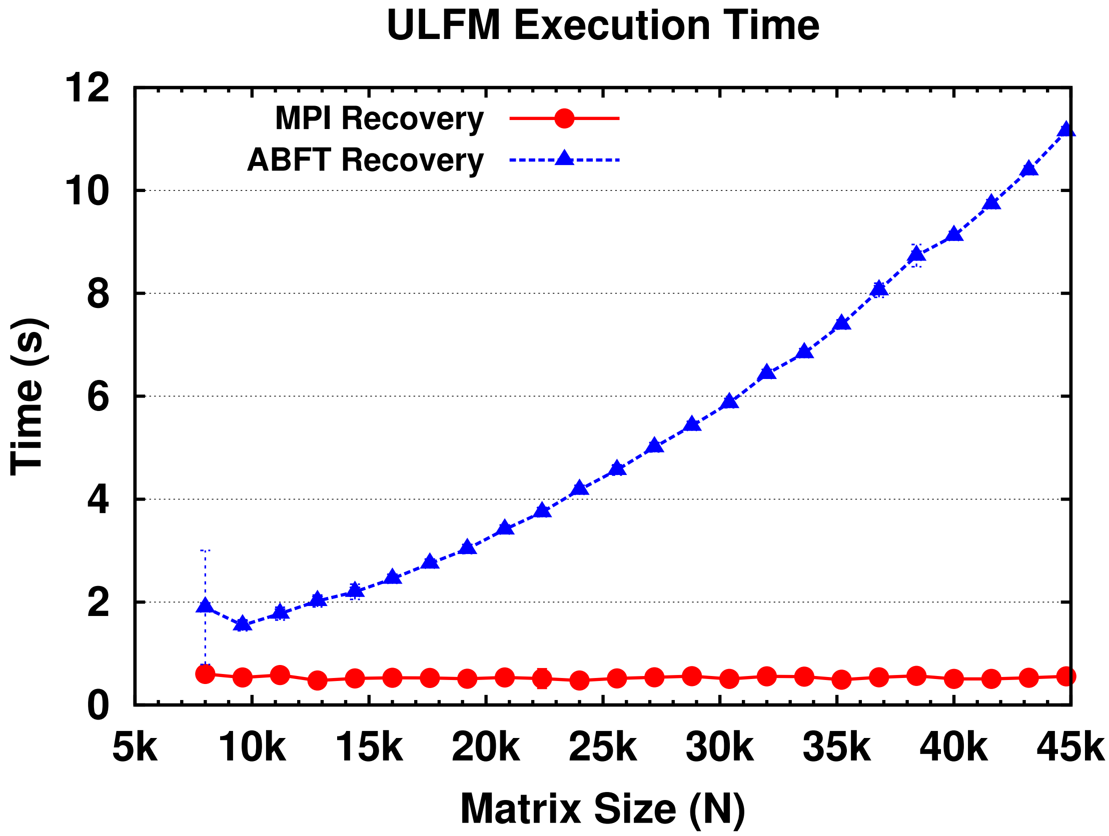


Figure 5.11: Recovery time of MPI and Data (via ABFT) on Grid5000

perform all of the recovery operations (`REVOKE`, `SHRINK`, `SPAWN`, `MERGE`, and `SPLIT`). The ABFT recovery time continues to scale with the problem size as the data recovery operations perform reductions across the entire matrix to calculate the missing data. Again these numbers are around the expected values and account for the disparity between the failure-free execution and the execution where a failure is injected early in the computation.

5.5 Evaluation of ULFM

Where CoF fell short of many of the goals in Chapter 3, ULFM is purposely designed to fulfill each of them. It provides the maximum amount of **flexibility** for fault tolerance because it is a minimal set of functions which provide a platform on which other types of fault tolerance can be constructed. It maintains **resilience** in the face of failures by ensuring that the library provides sufficient failure notification to prevent deadlock and introduces new constructs to allow the application or library to introduce more consistency when necessary. While **productivity** is not the strength of ULFM itself, it encourages new, portable libraries which will make the ideas of ULFM more available to non-experts who are not as familiar with the theory of fault tolerance. More information about this can be found in Chapter 6.

Chapter 6

Fault Tolerant Applications and Libraries

During the ULFM design process, the specific intention has been to not promote one form of fault tolerance over another. The primary reason for this is because, to this point, no type of fault tolerance has emerged as a single solution to all applications and this situation is not expected to change in the future. Applications will always need to evaluate their execution method and choose the type of fault tolerance which best fits.

To this end, ULFM was designed to support all types of fault tolerance by providing a high-performing, portable interface. One of the biggest barriers to entry in the current field of fault tolerance is the lack of portability for fault tolerant solutions, specifically those which involve MPI. No MPI implementation has become a de facto standard for fault tolerance and therefore none has not been adopted into the MPI Standard itself. It is our hope that this work will eventually provide that foundation upon with other solutions can build. While ULFM can be a fault tolerance solution for some applications, the end goal of this work is to encourage other developers to create libraries which implement both established and new types of fault tolerance using the mechanisms provided.

This section will explore how fault tolerance can be implemented with ULFM, both from an application perspective, and how libraries could be constructed using the constructs provided.

6.1 Types of Fault Tolerance

First we will evaluate the fault tolerance methods currently used in the research community and how they can be re-implemented using ULFM as the foundation for the MPI communication.

6.1.1 Automatic Methods

Despite the development of new forms of recovery with the potential to replace it, checkpoint/restart has remained a staple of fault tolerance. This is primarily due to the fact that it is already ubiquitous, and it is simple to understand and use. Because of all this, there is no reason to believe that the use of checkpoint/restart is likely to diminish in the near future.

ULFM makes bringing synchronous checkpoint/restart into the MPI application simple. An example of this is CoF as discussed in Chapter 4. CoF uses small checkpoints, but vastly improves the restart time because it does not require the application to re-enter the batch queue system. ULFM improves this scenario even more as it no longer requires most of the processes in the application to even restart. Instead, the application can roll back any processes which need to recover data, repair any communication objects in use, and continue with the existing MPI infrastructure.

Asynchronous checkpointing can also be added to ULFM as an external library. Message logging can be implemented by using existing PMPI (MPI standard profiling interface) hooks to capture messages as they are sent and received. To recover, a library can provide a function which simplifies the process of spawning replacement processes, replaying messages to the new processes using the locally logged messages, and continuing the normal execution.

To implement replication and migration with ULFM, again, the library would use PMPI hooks to capture messages between processes. This time, rather than logging the content of messages, the library would redirect messages to the appropriate processes in the case where they have been moved from their original rank. When the application (or some separate failure detector) detects a failure (or imminent failure), it can checkpoint the application, move it to a new processor, and restart it on the remote machine.

6.1.2 Algorithm Based Fault Tolerance

While ABFT describes a wide range of algorithms, ULFM has been uniquely designed to support them. Many ABFT algorithms do not require that all processes which begin an application remain running to completion. An example of such a class of applications is a Monte-Carlo master/worker application where a master, or group of master processes, divide and distribute work to a pool of worker processes. If a process in the worker pool fails, the worker does not need to be replaced. Only the work needs to be recovered, and it is given to another worker to complete in its place. For these types of applications, ULFM can often support them directly by providing the simple tool, `MPI_COMM_AGREE`. When a master process detects a failure, it removes the process from its internal list of alive workers (possibly informing other masters if they exist) and continues without any other MPI recover. When the application is ready to complete, the group of workers can call `MPI_COMM_AGREE` to determine if all of the master processes agree that they are finished or need to perform some other recovery.

For applications which require all processes to continue running through the application's completion, ULFM again provides all of the tools necessary. Upon failure, the application should call `MPI_COMM_REVOKE` to inform all other processes about the process failure, then the processes collectively call MPI_COMM_SHRINK to generate a working communicator without the failed processes. Next, the processes call the existing MPI function MPI_COMM_SPAWN to replace any failed processes with new ones. MPI_INTERCOMM_MERGE will create a more traditional intracommunicator from the intercommunicator generated by MPI_COMM_SPAWN. If the original ranks were important, the application can use MPI_COMM_SPLIT where all processes contribute the same color to signify that they will all remain in the same communicator and contribute their desired rank to the "key" value. At this point, the application is ready to repair any lost data and continue. These functions can be combined into a convenience function to simplify development, but the construction of an entirely new library is unnecessary for most forms of ABFT.

6.1.3 Transactional Fault Tolerance

Transactional fault tolerance is similar to the rollback recovery methods found in checkpoint/restart protocols. However, it also implies more automatic recovery than is provided in checkpoint/restart. Transactions can be constructed by adding a new mechanism to expand the functionality of MPI_COMM_AGREE. In addition to the agreement algorithm, the new function can store the state of the running application when the agreement algorithm determines that no failures occurred in the previous transaction, or it can roll the application back to a known good state when the previous transaction fails. In addition to rolling the existing processes back to a previous state, the library can perform the recovery methods described in Section 6.1.2 to restore any failed ranks using the existing data from the previous transaction.

6.1.4 Collective Consistency

One of the design decisions made when envisioning ULFM was to have failure knowledge be local. Failure notification on one process is no guarantee that any other processes are also aware of the failure. This decision was reached for performance reasons, however some applications may be willing to pay this performance cost in exchange for global knowledge of failures. For these applications, a library can easily be constructed to include collective consistency using the tools provided in ULFM. The goal of collective consistency is to ensure that all processes involved in a communication operation return an error code uniformly. To do this, a library can add a call to `MPI_COMM_AGREE` after the completion of each communication function which decides the status of previous operations. If any process returned a failure, then all remaining processes can agree on the return code and provide the same value upon exit. This allows the application to ignore the implications of local failure notification and perform recovery accordingly.

6.2 Library Construction

Given the emphasis laid on the ability to construct many varieties of fault tolerance using the tools provided by ULFM, one of the most important demonstrations to be made should be properly constructing libraries. The technique to do so was not as immediately apparent as it may seem so we detail it here to simplify the process in the future. This is not the only technique to properly construct a fault tolerant library on top of ULFM, but it can be used as a starting point for future work. More details, including a complete code example can be found in Appendix B.

6.2.1 Initialization

As with many scientific libraries, before using a library built on ULFM, it is advisable to create an initialization function. In addition to any usual data initialization which may occur during this time, this is also where any sub-communicators can be created. It is important to not base any communicators on `MPI_COMM_WORLD` as this communicator will become broken and out of date immediately following the first recovery or dynamic processing operation. Once a process has failed, there is no way to repair MPI_COMM_WORLD to its original state or to include any new processes which may be spawned to replace the failed processes. To solve this problem, applications should provide another communicator, possibly even a simple duplicate of MPI_COMM_WORLD, into the library through the initialization function so that sub-communicators can be constructed from this communicator, rather than MPI_COMM_WORLD, as has become a standard practice in many MPI libraries.

6.2.2 Status Object

Though not required, a status object can greatly simplify recovery later in an application. The status object can store useful pieces of data to be passed back and forth around library functions, but for the purposes of fault tolerance, the status object keeps track of the status of the most recent function calls. When a function is called, the object is passed into the function and the status of the function is updated throughout its execution. If a failure occurs and data is being recovered, the library can refer to the status object to discover what kinds of data to recover and signal the function that the library has been repaired. A status object is stored in the space of the calling application or library rather than within the library itself. The reason for this is so the status object may remain easily savable for fault tolerance, either by checkpointing, storage on a remote node, or duplication.

6.2.3 The Three R's

When a failure does occur, a fault tolerant library using ULFM should perform "Three R's" to get the library back into a functional state.

Revoke

First, the library should call MPI_COMM_REVOKE on all internal communicators to ensure that all other processes are alerted to the process failure. As most of the communicators will be reconstructed when the library is later being repaired anyway, this step does not introduce a level of overhead which would otherwise not have been present. Once all communicators have been revoked, it is safe to return from the library.

Return

The low level libraries should not attempt to perform process recovery automatically. The reason for this is that libraries generally do not make their internal communicators available to outside entities. If a library were to repair its own communicators by creating new processes to replace any failures, other libraries or parts of the application would no longer have access to these new processes as they would not be able to communicate through any existing channels. While it would be possible to create new communicators to solve this problem, the complexity introduced would not justify the effort and invalidate the convenience of performing the automatic recovery in the first place. In addition, the act of spawning new processes requires access to the original command line parameters. While these could be passed into the library to facilitate recovery, it is simpler to perform all of the actions at the same level, from the original application.

Repair

Once the libraries have revoked their internal communicators and returned to the highest level, the MPI recovery can begin. This should be a collaborative process between the application and all of the lower level libraries, however it should start with the application repairing MPI first. Depending on the application, this repair operation could include spawning new processes to replace any failures, or it could simply be calling `MPI_COMM_SHRINK` to remove any failed processes from the communicators. Once the application has repaired MPI, it should allow the libraries to repair themselves by providing the new MPI communicator to their repair functions. If the repair function will also repair any missing or corrupt data, the

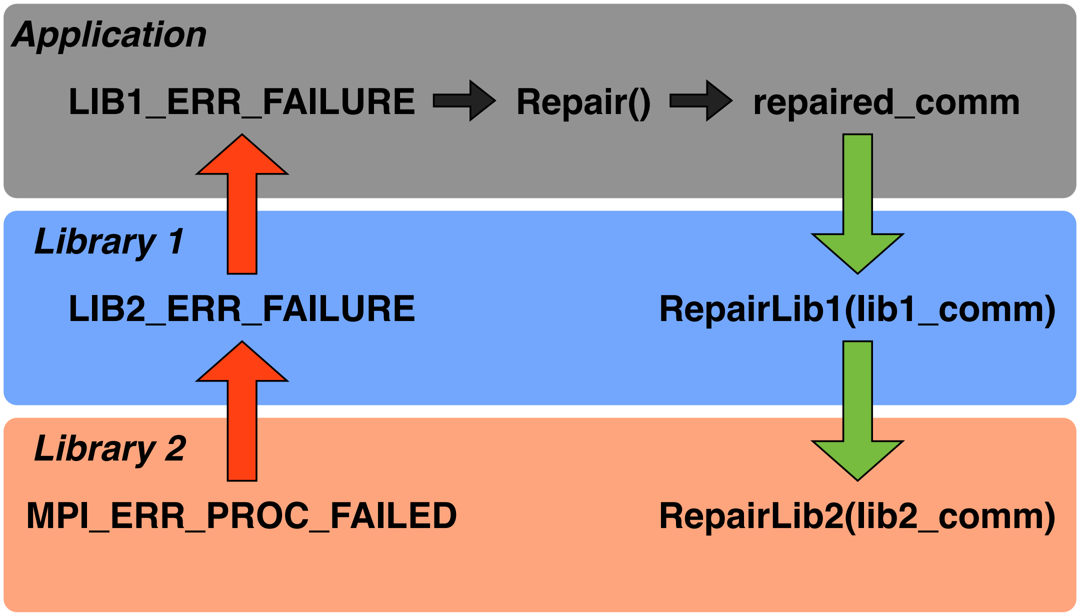


Figure 6.1: ABFT QR and one CoF recovery on Kraken (Lustre).

status object should also be included so the libraries will know the status of their previous operations and can recover accordingly. The libraries should continue to call any lower level repair functions for libraries on which they depend until all libraries have been appropriately repaired.

Overview

Figure 6.1 demonstrates the hierarchy of libraries and how they should be repaired. Errors will most likely be detected by the lowest level library currently in use. The libraries should recursively revoke their communicators and return to the next level. The application should repair the MPI library using the appropriate measures and allow the libraries to do the same by calling their respective repair functions. Once all of the recovery is complete, the application can repeat its call to the last function it was attempting and execution can continue.

Obviously, this pattern will not apply to all libraries. Some libraries developed to provide MPI fault tolerance directly may perform recovery themselves without returning to the MPI application. Some libraries may not include MPI calls which would necessitate recovery. However, this is a starting point for those interested in constructing fault tolerant MPI libraries. For a more complete example, see Appendix B.

Chapter 7

Future Work and Conclusions

7.1 Summary

As machine sizes and problem runtimes have increased over the decades, the rise of fault tolerance as a field of study has increased to match. Early on, applications developed methods of error checking and recovery to prevent faults from causing inconsistent results. Later, as the types of machines on which applications were being run evolved from large mainframe types of machines to Networks of Workstations (NoWs), checkpointing became important. Because workstations were considered unreliable as they could quickly become unavailable due either local use, or more common failures due to cheaper hardware, applications needed to be able to save their state during execution and possibly migrate from one machine to another. This started as a transparent feature that automatically performed checkpointing and migration and transitioned into a sophisticated system which could be triggered on-demand by an application, even performing asynchronous checkpoints which could later be used, along with message logging, to roll back applications to previous states.

All of these methods of fault tolerance were sufficient for the machines on which they were designed to function. The scale of the machines did not cause contention for bandwidth to stable storage, and failures did not occur with enough frequency to eclipse the time needed to perform a checkpoint. In recent years and going forward to projected machine architectures in the near future, these statements will not remain true. Machine sizes have already eclipsed the million core mark and runtimes for such large scale, capability applications extend to multiple days.

To solve this problem, new codes using Algorithm Based Fault Tolerance (ABFT) are now being designed which can repair themselves with very little data necessary. These algorithms have been proven to be effective and numerically stable, but to continue their parallel execution, they require a Message Passing Interface (MPI) library which can consistently provide communication channels, even after a process failure makes some subset of the machine unusable.

As a first step to provide the desired MPI implementation, we developed a new protocol called Checkpoint-on-Failure (CoF). This protocol provides an opportunity for applications to save their state *after* the application has detected a process failure. By changing the default MPI Error Handler from `MPI_ERRORS_ARE_FATAL` to `MPI_ERRORS_RETURN` or another custom error handler, the application is alerted to process failures and can incur the overhead of saving state only when process failures actually occur, rather than periodically throughout the application execution. In Chapter 4, we demonstrated the low overhead and recovery time that CoF provides.

Once the foundational work, such as a resilient runtime, was completed in the CoF implementation, we introduced a more ambitious project. User Level Failure Mitigation (ULFM), is a new chapter for the MPI Standard which provides a complete solution for fault tolerance, not just an improved checkpoint/restart protocol. ULFM allows ABFT codes to continue execution on all non-failed processes and replace failed processes with new ones which can be joined with already existing processes using (already standardized) MPI-2 dynamic processing functions. It does this by providing a minimal interface which includes failure detection, failure notification, and deadlock resolution mechanisms, while encouraging the development of new libraries to envision more complex recovery mechanisms, such as transactions, collective consistency, or automatic recovery.

7.2 Future Work

The tools developed in this work are extensive and sufficient for many styles of fault tolerance. However, they are not simple enough for developers not familiar with fault tolerance methods to construct complex recovery mechanisms. For this work to continue to be successful, more libraries will need to follow to provide interfaces which make fault tolerance more accessible.

One of the greatest challenges currently faced by researchers in the field of fault tolerance is apathy from those who they attempt to convince to adopt new fault tolerance techniques. For many years, scientists who develop codes for high performance architectures have been warned about the impending need for fault tolerance and the requirement that their codes be refactored to implement new protocols. However, the problems were largely resolved by implementing new automatic fault tolerance solutions, such as checkpoint/restart, which did not require that existing codes be modified, only that they be recompiled to include a new library.

Now, as new projections demonstrate the need for new methods of fault tolerance rather than an improved automatic solution [16], the need is not to convince developers to refactor their existing scientific codes to include fault tolerance, but to first convince researchers to develop easy to use, portable libraries which simplify the process of including fault tolerance in existing codes and provide resilience options for new codes being developed. These libraries will be much more adoptable and will speed the inclusion of fault tolerance in codes which already have expressed a need for such tools.

Bibliography

Bibliography

[1] Global Arrays. [Online]. Available: http://www.emsl.pnl.gov/docs/global/armci/ 11

[2] Intel MPI Benchmarks. [Online]. Available: http://software.intel.com/en-us/articles/intel-mpi-benchmarks 63

[3] June, 2011 Top500 List. [Online]. Available: http://www.top500.org/lists/2011/06/ 25

[4] MPICH. [Online]. Available: http://www.mpich.org 9

[5] MVAPICH2. [Online]. Available: http://mvapich.cse.ohio-state.edu/overview/mvapich2/ 9

[6] NetPIPE. [Online]. Available: http://www.scl.ameslab.gov/netpipe/ 63

[7] November, 2012 Top500 List. [Online]. Available: http://www.top500.org/lists/2012/11/ 1

[8] Open MPI. [Online]. Available: http://www.open-mpi.org/ 9, 25

[9] Sequoia AMG Benchmarks. [Online]. Available: https://asc.llnl.gov/sequoia/benchmarks/ 64

[10] A. M. Agbaria and R. Friedman, "Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations," *High Performance Distributed Computing,*

1999. Proceedings. The Eighth International Symposium on, pp. 167–176, 1999. 24

[11] E. Anderson, Z. Bai, J. DONGARRA, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, "LAPACK: a portable linear algebra library for high-performance computers," in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing.* IEEE Computer Society Press, Oct. 1990. 38

[12] R. Batchu, Y. S. Dandass, A. Skjellum, and M. Beddhu, "MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware," *Cluster Computing*, vol. 7, no. 4, pp. 303–315, 2004. 23

[13] P. A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys (CSUR)*, vol. 13, no. 2, pp. 185–221, 1981. 19

[14] M. Bhandarkar, L. V. Kale, E. de Sturler, and J. Hoeflinger, "Object-based adaptive load balancing for MPI programs," in *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, May 2001, pp. 108–117. 9, 10

[15] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. S. A. I. . C. Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes," in *Supercomputing, ACM/IEEE 2002 Conference.* 22

[16] G. Bosilca, A. Bouteiller, É. Brunet, F. Cappello, J. Dongarra, A. Guermouche, T. Herault, Y. Robert, F. Vivien, and D. Zaidouni, "Unified Model for Assessing Checkpointing Protocols at Extreme-Scale," Innovative Computing Laboratory, Département Informatique, GRAND-LARGE - INRIA Saclay - Ile de France, Joint Laboratory for Petascale Computing, Laboratoire de Recherche

en Informatique, ROMA - ENS Lyon / CNRS / Inria Grenoble Rhône-Alpes, Laboratoire de l'Informatique du Parallélisme, Tech. Rep. RR-7950, May 2012. 17, 82

[17] A. Bouteiller, G. Bosilca, and J. Dongarra, "Redesigning the message logging model for high performance," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 16, pp. 2196–2211, 2010. 16

[18] P. Bridges, N. Doss, W. Gropp, E. Karrels, E. Lusk, and A. Skjellum, "Users' Guide to mpich, a Portable Implementation of MPI," Mathematics and Computer Science Division, Argonne National Laboratory, Chicago, IL, Tech. Rep., 1995. 9

[19] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," in *Proceedings of Supercomputing Symposium*, 1994, pp. 379–386. 9

[20] D. Callahan, B. L. Chamberlain, and H. P. Zima, "The cascade high productivity language," in *High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings. Ninth International Workshop on*, 2004, pp. 52–60. 10

[21] F. Cappello, "Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities," *International Journal of High Performance Computing Applications*, vol. 23, no. 3, pp. 212–226, Jul. 2009. 1

[22] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," Tech. Rep. CCS-TR-99-157, May 1999. 11

[23] S. Chakravorty, C. Mendes, and L. Kalé, "Proactive fault tolerance in MPI applications via task migration," *High Performance Computing-HiPC 2006*, pp. 485–496, 2006. 25

[24] K. M. Chandy and L. Lamport, "Distributed snapshots : Determining global states of distributed systems," in *Transactions on Computer Systems*, vol. 3(1). ACM, February 1985, pp. 63–75. 13

[25] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra, "Building fault survivable MPI programs with FT-MPI using diskless checkpointing," in *Proceedings for ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2005, pp. 213–223. 19

[26] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, and D. Walker, "A proposal for a set of parallel basic linear algebra subprograms," Knoxville, TN, USA, Tech. Rep., 1995. 38

[27] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, "ScaLAPACK: A portable linear algebra library for distributed memory computers—Design issues and performance," *Computer physics communications*, vol. 97, no. 1, pp. 1–15, 1996. 38

[28] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *ACM Transactions on Mathematical Software*, vol. 16, pp. 1–17, Mar. 1990. 36

[29] J. J. Dongarra and R. A. Geijn, "Two Dimensional Basic Linear Algebra Communication Subprograms," Computer Science Department, University of Tennessee, Knoxville, TN, Tech. Rep. UT CS-91-138, Oct. 1991. 38

[30] J. J. Dongarra and R. C. Whaley, "A User's Guide to the BLACS v1.0," Computer Science Department, University of Tennessee, Knoxville, TN, Tech. Rep. UT CS-95-281, Mar. 1995. 38, 60

[31] J. Dongarra, L. Blackford, J. Choi *et al.*, "ScaLAPACK user's guide," *Society for Industrial and Applied Mathematics, Philadelphia, PA*, 1997. 36

[32] P. Du, A. Bouteiller *et al.*, “Algorithm-based fault tolerance for dense matrix factorizations,” in *17th SIGPLAN PPoPP*. ACM, 2012, pp. 225–234. xii, 36, 37, 42

[33] J. Duell, “The Design and Implementation of Berkeley Lab’s Linux Checkpoint/Restart,” Tech. Rep. LBNL-54941, December 2002. 15, 16

[34] G. Fagg and J. Dongarra, “FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world,” *EuroPVM/MPI*, 2000. 9, 20, 51

[35] G. E. Fagg, A. Bukovsky, and J. J. Dongarra, “HARNESS and fault tolerant MPI,” *Parallel Computing*, vol. 27, no. 11, pp. 1479–1495, 2001. 21

[36] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold, “Evaluating the viability of process replication reliability for exascale systems,” p. 44, 2011. 17

[37] M. Fischer and N. Lynch, “Impossibility of distributed consensus with one faulty process,” *Journal of the ACM (JACM)*, 1985. 33

[38] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, and A. Lumsdaine, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pp. 353–377, 2004. 9, 25

[39] M. Herlihy, J. Eliot, and B. Moss, “Transactional Memory: Architectural Support For Lock-free Data Structures,” in *Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on*. 19

[40] K.-H. Huang and J. Abraham, “Algorithm-Based Fault Tolerance for Matrix Operations,” *IEEE Transactions on Computers*, no. 6, pp. 518–528, 1984. 18

[41] W. Huang, G. Santhanaraman, H. W. Jin, Q. Gao, and D. K. Panda, "Design of High Performance MVAPICH2: MPI2 over InfiniBand," *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, vol. 1. 9

[42] J. Hursey, T. Naughton, G. Vallee, and R. L. Graham, "A log-scaling fault tolerant agreement algorithm for a fault tolerant MPI," in *Recent Advances in the Message Passing Interface*, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. S. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2011, vol. 6960, pp. 255–263. 47

[43] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The design and implementation of checkpoint/restart process fault tolerance for Open MPI," *International Parallel and Distributed Processing Symposium (IPDPS 2007)*, pp. 1–8, 2007. 22

[44] L. V. Kale and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based On C++," in *In Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applicaions*, 1993, pp. 91–108. 10

[45] S. Kulkarni and A. Arora, "Automating the addition of fault-tolerance," *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pp. 339–359, 2000. 8

[46] M. Litzkow and M. Solomon, "Supporting checkpointing and process migration outside the UNIX kernel," in *Usenix Winter Conference.* San Francisco, CA: Citeseer, 1992. 15, 16

[47] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, "Checkpoint and migration of UNIX processes in the Condor distributed processing system," Tech. Rep. 1346, Apr. 1997. 15

[48] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory

programming toolkit," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006. 11

[49] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under Unix," in *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings.* USENIX Association, Jan. 1995. 14, 18

[50] J. Plank, Y. Kim, and J. Dongarra, "Algorithm-based diskless checkpointing for fault tolerant matrix operations," *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pp. 351–360, 1995. 18

[51] S. Rao, L. Alvisi, and H. M. Vin, "Egida: An extensible toolkit for low-overhead fault-tolerance," *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pp. 48–55, 1999. 16, 24

[52] H. Richardson, "High performance fortran: history, overview and current developments," 1.4 TMC-261, Thinking Machines Corporation, Tech. Rep., 1996. 11

[53] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems," *IEEE International Parallel and Distributed Processing Symposium, 2007. IPDPS 2007.*, pp. 1–10, 2007. 24

[54] S. Sankaran, "The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing," *International Journal of High Performance Computing Applications*, vol. 19, no. 4, pp. 479–493, Nov. 2005. 21

[55] B. Schroeder and G. A. Gibson, "Understanding Failures in Petascale Computers," *SciDAC, Journal of Physics: Conference Series*, vol. 78, 2007. 1

[56] J. M. Squyres and A. Lumsdaine, "A Component Architecture for LAM/MPI," in *Proceedings, 10th European PVM/MPI Users' Group Meeting*, ser. Lecture

Notes in Computer Science, no. 2840. Venice, Italy: Springer-Verlag, September / October 2003, pp. 379–387. 9

[57] V. S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: practice and experience*, vol. 2, no. 4, pp. 315–339, Dec. 1990. 9

[58] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: The Condor experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 24, pp. 323–356, 2005. 2

[59] The MPI Forum, "MPI: A Message-Passing Interface Standard, Version 3.0," Tech. Rep., 2012. 8, 30

[60] J. P. Walters and V. Chaudhary, "Replication-Based Fault Tolerance for MPI Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 7, pp. 997–1010. 21

[61] Y.-M. Wang, Y. Huang, K.-P. Vo, P. Chung, and C. Kintala, "Checkpointing and its applications," *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pp. 22–31, 1995. 14, 16

Appendices

Appendix A

Process Fault Tolerance

Chapter Submitted to MPI Forum. Section references not preceded by an `A` refer to the MPI 3.0 Standard document.

A.1 Introduction

Long running and large scale applications are at increased risk of encountering process failures during normal execution. We consider a process failure as a fail-stop failure; failed processes become permanently unresponsive to communications. This chapter introduces the MPI features that support the development of applications and libraries that can tolerate process failures. The approach described in this chapter is intended to prevent the deadlock of processes while avoiding impact on the failure-free execution of an application.

The expected behavior of MPI in the case of a process failure is defined by the following statements: any MPI operation that involves a failed process must not block indefinitely, but either succeed or raise an MPI exception (see Section A.2); an MPI operation that does not involve the failed process will complete normally, unless interrupted by the user through provided functionality. Asynchronous failure propagation is not required. If an application needs global knowledge of failures, it can use the interfaces defined in Section A.3 to explicitly propagate locally detected failures.

An implementation that does not tolerate process failures must provide the interfaces and semantics defined in this chapter as long as no failure occurred. It must never raise an exception of class MPI_ERR_PROC_FAILED or MPI_ERR_PENDING because of a process failure. This chapter does not define process failure semantics for the operations specified in Chapters 10, 11 and 12, therefore they remain undefined by the MPI standard.

Advice to Users Many of the operations and semantics described in this chapter are only applicable when the MPI application has replaced the default error handler MPI_ERRORS_ARE_FATAL on, at least, MPI_COMM_WORLD.

A.2 Failure Notification

This section specifies the behavior of an MPI communication operation when failures occur on processes involved in the communication. A process is considered involved in a communication if any of the following is true:

1. the operation is collective and the process appears in one of the groups of the associated communication object;
2. the process is a specified or matched destination or source in a point-to-point communication;
3. the operation is an MPI_ANY_SOURCE receive operation and the failed process belongs to the source group.

Therefore, if an operation does not involve a failed process (such as a point-to-point message between two non-failed processes), it must not raise a process failure exception.

Advice to Implementors A correct MPI implementation may provide failure detection only for processes involved in an ongoing operation, and postpone detection of other failures until necessary. Moreover, as long as an implementation can complete operations, it may choose to delay raising an error. Another valid implementation might choose to raise an error as quickly as possible.

Non-blocking operations must not raise an exception about process failures during initiation. All process failure errors are postponed until the corresponding completion function is called.

A.2.1 Startup and Finalize

Advice to Implementors If a process fails during MPI_INIT but its peers are able to complete the MPI_INIT successfully, then a high quality implementation will return MPI_SUCCESS and delay the reporting of the process failure to a subsequent MPI operation.

MPI_FINALIZE will complete successfully even in the presence of process failures.

Advice to Users Considering Example 8.7 in Section 8.7, the process with rank 0 in MPI_COMM_WORLD may have failed before, during, or after the call to MPI_FINALIZE. MPI only provides failure detection capabilities up to when MPI_FINALIZE is invoked and provides no support for fault tolerance during or after MPI_FINALIZE. Applications are encouraged to implement all rank-specific code before the call to MPI_FINALIZE to handle the case where process 0 in MPI_COMM_WORLD fails.

A.2.2 Point-to-Point and Collective Communication

An MPI implementation raises the following error classes to notify users that a point-to-point communication operation could not complete successfully because of the failure of involved processes:

- MPI_ERR_PENDING indicates, for a non-blocking communication, that the communication is a receive operation from MPI_ANY_SOURCE and no matching send has been posted, yet a potential sender process has failed. Neither the operation nor the request identifying the operation are completed. Note that the same error class is also used in status when another communication raises an exception during the same operation (as defined in Section 3.7.5).
- In all other cases, the operation raises an exception of class MPI_ERR_PROC_FAILED to indicate that the failure prevents the operation from following its failure-free specification. If there is a request identifying the point-to-point communication, it is completed. Future point-to-point communication with the same process on this communicator must also raise MPI_ERR_PROC_FAILED.

Advice to Users To acknowledge a failure and discover which processes failed, the user should call MPI_COMM_FAILURE_ACK (as defined in Section A.3.1).

When a collective operation cannot be completed because of the failure of an involved process, the collective operation raises an error of class MPI_ERR_PROC_FAILED.

Advice to Users Depending on how the collective operation is implemented and when a process failure occurs, some participating alive processes may raise an exception while other processes return successfully from the same collective operation. For example, in MPI_BCAST, the root process may succeed before a failed process disrupts the operation, resulting in some other processes raising an error. However, it is noteworthy that for collective operations on an intracommunicator in which all processes contribute to the result and all processes receive the result, processes which do not enter the operation due to process failure provoke all surviving ranks to raise MPI_ERR_PROC_FAILED. Similarly, for the same collective operations on an intercommunicator, a process in the remote group which failed before entering the operation has the same effect on all surviving ranks of the local group.

Advice to Users Note that communicator creation functions (like MPI_COMM_DUP or MPI_COMM_SPLIT) are collective operations. As such, if a failure happened during the call, an error might be raised at some processes while others succeed and obtain a new communicator. While it is valid to communicate between processes which succeeded to create the new communicator, it is the responsibility of the user to ensure that all involved processes have a consistent view of the communicator creation, if needed. A conservative solution is to have each process either revoke (see Section A.3.1) the parent communicator if the operation fails, or call an MPI_BARRIER on the parent communicator and then revoke the new communicator if the MPI_BARRIER fails.

When a communication operation raises an exception related to process failure, the content of the output buffers is *undefined*.

A.2.3 Dynamic Process Management

Dynamic process management functions require some additional semantics from the MPI implementation as detailed below.

1. If the MPI implementation raises an error related to process failure to the root process of MPI_COMM_CONNECT or MPI_COMM_ACCEPT, at least the root processes of both intracommunicators must raise the same error of class MPI_ERR_PROC_FAILED (unless required to raise MPI_ERR_REVOKED as defined by A.3.1).

2. If the MPI implementation raises an error related to process failure to the root process of MPI_COMM_SPAWN, no spawned processes should be able to communicate on the created intercommunicator.

Advice to Users As with communicator creation functions, it is possible that if a failure happens during dynamic process management operations, an error might be raised at some processes while others succeed and obtain a new communicator.

A.2.4 One-Sided Communication

As with all nonblocking operations, one-sided communication operations should delay all failure notification until their synchronization operations which may raise MPI_ERR_PROC_FAILED (see Section A.2). If the implementation raises an error related to process failure from the synchronization function, the epoch behavior is unchanged from the definitions in Section 11.4. As with collective operations over MPI communicators, it is possible that some processes have detected a failure and raised MPI_ERR_PROC_FAILED, while others returned MPI_SUCCESS.

Unless specified below, the state of memory targeted by any process in an epoch in which operations raised an error related to process failure is undefined.

1. If a failure is to be reported during active target communication functions MPI_WIN_COMPLETE or MPI_WIN_WAIT (or the non-blocking equivalent MPI_WIN_TEST), the epoch is considered completed and all operations not involving the failed processes must complete successfully.

2. If the target rank has failed, MPI_WIN_LOCK and MPI_WIN_UNLOCK operations raise an error of class MPI_ERR_PROC_FAILED. If the owner of a lock has failed, the lock cannot be acquired again, and all subsequent operations on the lock must raise MPI_ERR_PROC_FAILED.

Advice to Users It is possible that request-based RMA operations complete successfully while the enclosing epoch completes by raising error due to process failure. In this scenario, the local buffer is valid but the remote targeted memory is undefined.

A.2.5 I/O

I/O error classes and their consequences are defined in Section 13.7. The following section defines the behavior of I/O operations when MPI process failures prevent their successful completion. Since collective I/O operations may not synchronize with other processes, process failures may not be reported during a collective I/O operation. If a process failure prevents a file operation from completing, an MPI exception of class `MPI_ERR_PROC_FAILED` is raised. Once an MPI implementation has raised an error of class `MPI_ERR_PROC_FAILED`, the state of the file pointer is *undefined*.

Advice to Users Users are encouraged to use `MPI_COMM_AGREE` on a communicator containing the same group as the file handle, to deduce the completion status of collective operations on file handles and maintain a consistent view of file pointers.

A.3 Failure Mitigation Functions

A.3.1 Communicator Functions

MPI provides no guarantee of global knowledge of a process failure. Only processes involved in a communication operation with the failed process are guaranteed to eventually detect its failure (see Section A.2). If global knowledge is required, MPI provides a function to revoke a communicator at all members.

MPI_COMM_REVOKE(comm)

`IN`	`comm`	**communicator (handle)**

This function notifies all processes in the groups (local and remote) associated with the communicator *comm* that this communicator is now considered revoked. This function is not collective and therefore does not have a matching call on remote processes. It is erroneous to call `MPI_COMM_REVOKE` on a communicator for which no operation raised an MPI exception related to process failure. All alive processes belonging to *comm* will be notified of the revocation despite failures. The revocation of a communicator completes any non-local MPI operations on *comm* by raising an error of class `MPI_ERR_REVOKED`, with the exception of `MPI_COMM_SHRINK` and

MPI_COMM_AGREE (and its nonblocking equivalent). A communicator becomes revoked as soon as:

1. MPI_COMM_REVOKE is locally called on it;
2. Any MPI operation raised an error of class MPI_ERR_REVOKED because another process in *comm* has called MPI_COMM_REVOKE.

Once a communicator has been revoked, all subsequent non-local operations on that communicator, with the exception of MPI_COMM_SHRINK and MPI_COMM_AGREE (and its nonblocking equivalent), are considered local and must complete by raising an error of class MPI_ERR_REVOKED.

Advice to Users High quality implementations are encouraged to do their best to free resources locally when the user calls free operations on revoked communication objects, or communication objects containing failed processes.

MPI_COMM_SHRINK(comm, newcomm)

IN	comm	**communicator (handle)**
OUT	newcomm	**communicator (handle)**

This collective operation creates a new intra or inter communicator *newcomm* from the revoked intra or inter communicator *comm* respectively by excluding its failed processes as detailed below. It is erroneous MPI code to call MPI_COMM_SHRINK on a communicator which has not been revoked (as defined above) and will raise an error of class MPI_ERR_ARG.

This function must not raise an error due to process failures (error classes MPI_ERR_PROC_FAILED and MPI_ERR_REVOKED). All processes that succeeded agreed on the content of the group of processes that failed. This group includes at least every process failure that has raised an MPI exception of class MPI_ERR_PROC_FAILED or MPI_ERR_PENDING. The call is semantically equivalent to an MPI_COMM_SPLIT operation that would succeed despite failures, and where living processes participate with the same color, and a key equal to their rank in *comm* and failed processes implicitly contribute MPI_UNDEFINED.

Advice to Users This call does not guarantee that all processes in *newcomm* are alive. Any new failure will be detected in subsequent MPI operations.

MPI_COMM_FAILURE_ACK(comm)

IN	comm	**communicator (handle)**

This local operation gives the users a way to *acknowledge* all locally notified failures on *comm*. After the call, unmatched MPI_ANY_SOURCE receptions that would have raised an error code MPI_ERR_PENDING due to process failure (see Section A.2.2) proceed without further reporting of errors due to those acknowledged failures.

Advice to Users Calling MPI_COMM_FAILURE_ACK on a communicator with failed processes does not allow that communicator to be used successfully for collective operations. Collective communication on a communicator with acknowledged failures will continue to raise an error of class MPI_ERR_PROC_FAILED as defined in Section A.2.2. To reliably use collective operations on a communicator with failed processes, the communicator should first be revoked using MPI_COMM_REVOKE and then a new communicator should be created using MPI_COMM_SHRINK.

MPI_COMM_FAILURE_GET_ACKED(comm, failedgroup)

IN	comm	**communicator (handle)**
OUT	failedgroup	**group (handle)**

This local operation returns the group *failedgrp* of processes, from the communicator *comm*, which have been locally acknowledged as failed by preceding calls to MPI_COMM_FAILURE_ACK. The failedgrp can be empty, that is, equal to MPI_GROUP_EMPTY.

MPI_COMM_AGREE(comm, flag)

`IN`	`comm`	**communicator (handle)**
`INOUT`	`flag`	**boolean flag**

This function performs a collective operation on the group of living processes in *comm*. On completion, all living processes must agree to set the output value of *flag* to the result of a logical *'AND'* operation over the input values of *flag*. This function must not raise an error due to process failure (error classes `MPI_ERR_PROC_FAILED` and `MPI_ERR_REVOKED`), and processes that failed before entering the call do not contribute to the operation.

If *comm* is an intercommunicator, the value of *flag* is a logical *'AND'* operation over the values contributed by the remote group (where failed processes do not contribute to the operation).

Advice to Users `MPI_COMM_AGREE` maintains its collective behavior even if the *comm* is revoked.

MPI_COMM_IAGREE(comm, flag, req)

`IN`	`comm`	**communicator (handle)**
`INOUT`	`flag`	**boolean flag**
`OUT`	`req`	**request (handle)**

This function has the same semantics as `MPI_COMM_AGREE` except that it is nonblocking.

A.3.2 One-Sided Functions

MPI_WIN_REVOKE (win)

`IN`	`win`	**window (handle)**

This function notifies all processes within the window *win* that this window is now considered revoked. A revoked window completes any non-local MPI operations on *win* with error and causes any new operations to complete with error. Once a window has been revoked, all subsequent non-local operations on that window are considered local and must fail with an error of class `MPI_ERR_REVOKED`.

MPI_WIN_GET_FAILED(win, failedgroup)

IN	win	**window (handle)**
OUT	failedgroup	**group (handle)**

This local operation returns the group *failedgrp* of processes from the window *win* which are locally known to have failed.

Advice to Users MPI makes no assumption about asynchronous progress of the failure detection. A valid MPI implementation may choose to only update the group of locally known failed processes when it enters a synchronization function.

Advice to Users It is possible that only the calling process has detected the reported failure. If global knowledge is necessary, processes detecting failures should use the call `MPI_WIN_REVOKED`.

A.3.3 I/O Functions

MPI_FILE_REVOKE (fh)

IN	fh	**file (handle)**

This function notifies all ranks within file *fh* that this file handle is now considered revoked.

Ongoing non-local completion operations on a revoked file handle raise an exception of class `MPI_ERR_REVOKED`. Once a file handle has been revoked, all subsequent non-local operations on the file handle must raise an MPI exception of class `MPI_ERR_REVOKED`.

A.4 Error Codes and Classes

The following error classes are added to those defined in Section 8.4:

Table A.1: Additional process fault tolerance error classes

MPI_ERR_PROC_FAILED	The operation could not complete because of a process failure (a fail-stop failure).
MPI_ERR_REVOKED	The communication object used in the operation has been revoked.

A.5 Examples

A.5.1 Master/Worker

The example below presents a master code that handles failures by ignoring failed processes and resubmitting requests. It demonstrates the different failure cases that may occur when posting receptions from MPI_ANY_SOURCE as discussed in the advice to users in Section A.2.2.

```
int master(void)
{
  MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN);
  MPI_Comm_size(comm, &size);

  /* .. submit the initial work requests .. */

  MPI_Irecv( buffer, 1, MPI_INT, MPI_ANY_SOURCE,
             tag, comm, &req );

  /* Progress engine: Get answers, send new requests,
     and handle process failures */
  while( (active_workers > 0) && work_available ) {
    rc = MPI_Wait( &req, &status );
    if( (MPI_ERR_PROC_FAILED == rc) ||
        (MPI_ERR_PENDING == rc) ) {
      MPI_Comm_failure_ack(comm);
      MPI_Comm_failure_get_acked(comm, &g);
      MPI_Group_size(g, &gsize);

      /* .. find the lost work and requeue it .. */

      active_workers = size - gsize - 1;
      MPI_Group_free(&g);

      /* repost the request if it
       * matched the failed process */
      if( rc == MPI_ERR_PROC_FAILED )
        MPI_Irecv( buffer, 1, MPI_INT, MPI_ANY_SOURCE,
                   tag, comm, &req );
      }

      continue;
    }

    /* .. process the answer and update work_available .. */
    MPI_Irecv( buffer, 1, MPI_INT, MPI_ANY_SOURCE,
               tag, comm, &req );
  }

  /* .. cancel request and cleanup .. */
}
```

A.5.2 Iterative Refinement

The example below demonstrates a method of fault-tolerance to detect and handle failures. At each iteration, the algorithm checks the return code of the MPI_-ALLREDUCE. If the return code indicates a process failure for at least one process, the algorithm revokes the communicator, agrees on the presence of failures, and later shrinks it to create a new communicator. By calling MPI_COMM_REVOKE, the algorithm ensures that all processes will be notified of process failure and enter the MPI_COMM_AGREE. If a process fails, the algorithm must complete at least one more iteration to ensure a correct answer.

```
while( gnorm > epsilon ) {
  /* Add a computation iteration to converge and
     compute local norm in lnorm */
  rc = MPI_Allreduce( &lnorm, &gnorm, 1,
             MPI_DOUBLE, MPI_MAX, comm);

  if( (MPI_ERR_PROC_FAILED == rc ) ||
      (MPI_ERR_COMM_REVOKE == rc) ||
      (gnorm <= epsilon) ) {

    if( MPI_ERR_PROC_FAILED == rc )
        MPI_Comm_revoke(comm);

    /* About to leave: let's be sure that everybody
       received the same information */
    allsucceeded = (rc == MPI_SUCCESS);
    MPI_Comm_agree(comm, &allsucceeded);
    if( !allsucceeded ) {
      /* We plan to join the shrink, thus the
         communicator should be marked as revoked */
      MPI_Comm_revoke(comm);
      MPI_Comm_shrink(comm, &comm2);

      /* Release the revoked communicator */
      MPI_Comm_free(comm);
      comm = comm2;

      /* Force one more iteration */
      gnorm = epsilon + 1.0;
    }
  }
}
```

Appendix B

Library Composition

Library composition in fault tolerance is considered an especially difficult problem. To demonstrate the feasibility of our solution, this appendix includes a sample implementation of a hierarchy of libraries. This code demonstrates token libraries to scale and add two vectors. While the function of the libraries is not important, the initialization and recovery code within the libraries is the key contributed to be noted here. While this is certainly not the only possible implementation of a library and probably not even the most efficient, it is a good reference for developers as an example of how to construct their recovery mechanisms.

B.1 Main application

This is the main code of the application. When a failure occurs in one of the lower level libraries, they will return control to this library to perform high level recovery (including repairing the MPI communicators) and then call the repair functions for the low level libraries before returning to the partially completed function calls.

vector_math.c

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#include "lib1.h"
#include "lib2.h"

#include "mpi.h"
#include "mpi-ext.h"

void repair(float *v1, float *v2, float *result, int
   size_v, MPI_Comm broken, MPI_Comm* repaired);

char *filename;
enum progression {
    START,
    INIT_LIB1_V1,
    INIT_LIB1_V2,
    INIT_LIB2,
    SCALE1,
    SCALE2,
    ADD
};
enum progression alg_status = START;
lib1_status_t lib1_status1, lib1_status2;
lib2_status_t lib2_status;

int main(int argc, char **argv) {
```

```
FILE *input;
int rank, size;
MPI_Comm parent, *world, *repair_comm;
float *vector1, *vector2, *result, *final_total;
int size_vector, temp, ret, old_rank;
char *buffer, *next;
int i, done;
MPI_Status mpi_status;

if (argc < 3) {
    fprintf(stderr, "Usage: ./vector_math
       vector1_file vector2_file\n");
    exit(1);
}

world = (MPI_Comm *) malloc(sizeof(MPI_Comm));

filename = strdup(argv[1]);

buffer = (char *) malloc(sizeof(char) * 1024);

input = fopen(argv[1], "r");
buffer = fgets(buffer, 1024, input);

/* We don't do a ton of error checking here. Don't
   let the line length overflow the buffer. You can
   use as many lines as you like. Seperate floats by
   whitespace. */
```

```
size_vector = (int) strtol(buffer, &next, 10);
if (size_vector == 0 && buffer == next) {
    fprintf(stderr, "Invalid input file\n");
    exit(1);
}
buffer = next;
vector1 = (float *) calloc(size_vector,
   sizeof(float));
for (i = 0; i < size_vector; i++) {
    vector1[i] = (float) strtod(buffer, &next);
    if (buffer == next || NULL == next) {
        buffer = fgets(buffer, 1024, input);
        if (NULL == next) {
            i--;
        }
    } else {
        buffer = next;
    }
}

result = (float *) malloc(sizeof(float) *
   size_vector);
final_total = (float *) malloc(sizeof(float) *
   size_vector);

input = fopen(argv[2], "r");
buffer = fgets(buffer, 1024, input);
```

```
/* We don't do a ton of error checking here. Don't
   let the line length overflow the buffer. You can
   use as many lines as you like. Seperate floats by
   whitespace. */
temp = (int) strtol(buffer, &next, 10);
if (temp == 0 && buffer == next) {
    fprintf(stderr, "Invalid input file\n");
    exit(1);
}

if (temp != size_vector) {
    fprintf(stderr, "Vectors should be the same
       size\n");
    exit(1);
}
buffer = next;

vector2 = (float *) calloc(size_vector,
   sizeof(float));
for (i = 0; i < size_vector; i++) {
    vector2[i] = (float) strtod(buffer, &next);
    if (buffer == next || NULL == next) {
        buffer = fgets(buffer, 1024, input);
        if (NULL == next) {
            i--;
        }
    } else {
        buffer = next;
    }
}

MPI_Init(&argc, &argv);

MPI_Comm_get_parent(&parent);

/* This is not an original process, perform recovery
   */
if (MPI_COMM_NULL != parent) {
    /* For now, abort if there is an error. Trying to
       handle all of the cases for failure during
       startup is unnecessarily complicated. Just
       abort the new processes and start over if
       there's a problem. */
    MPI_Comm_set_errhandler(parent,
       MPI_ERRORS_ARE_FATAL);

    /* Join the rest of the processes */
    repair_comm = (MPI_Comm *)
       malloc(sizeof(MPI_Comm));
    MPI_Intercomm_merge(parent, true, repair_comm);

    MPI_Comm_free(&parent);

    MPI_Recv(&old_rank, 1, MPI_INT, 0, 31337,
       *repair_comm, &mpi_status);
```

```
MPI_Comm_split(*repair_comm, 0, old_rank, world);

MPI_Comm_rank(*world, &rank);
MPI_Comm_size(*world, &size);

size_vector /= (size - 1);

/* Now we will start recovering from failures as
   we have one big world again and it is possible
   to reason about the status of the comm. */
MPI_Comm_set_errhandler(*world,
   MPI_ERRORS_RETURN);

/* Figure out where we were before we died */
if (MPI_SUCCESS != (ret =
   MPI_Allreduce(&alg_status, &alg_status, 1,
   MPI_INT, MPI_MAX, *world))) {
    /* Perform recovery */
    OMPI_Comm_revoke(*world);
    repair(vector1, vector2, result, size_vector,
       *world, world);
}

/* Perform ABFT recovery */
if (INIT_LIB1_V1 >= alg_status) {
    if (MPI_SUCCESS != (ret =
       lib1_recovery(vector1, size_vector,
       *world, &lib1_status1, 0))) {
        /* Failure during the library, perform
           recovery */
        OMPI_Comm_revoke(*world);
        repair(vector1, vector2, result,
           size_vector, *world, world);
    }
} else {
    if (MPI_SUCCESS != (ret = lib1_init(*world,
       &lib1_status1))) {
        /* Failure during the library, perform
           recovery */
        OMPI_Comm_revoke(*world);
        repair(vector1, vector2, result,
           size_vector, *world, world);
    }
    alg_status = INIT_LIB1_V1;
}

/* Perform ABFT recovery */
if (INIT_LIB1_V2 >= alg_status) {
    if (MPI_SUCCESS != (ret =
       lib1_recovery(vector2, size_vector,
       *world, &lib1_status2, 0))) {
        /* Failure during the library, perform
           recovery */
        OMPI_Comm_revoke(*world);
        repair(vector1, vector2, result,
           size_vector, *world, world);
    }
} else {
    if (MPI_SUCCESS != (ret = lib1_init(*world,
       &lib1_status2))) {
        /* Failure during the library, perform
           recovery */
        OMPI_Comm_revoke(*world);
        repair(vector1, vector2, result,
           size_vector, *world, world);
    }
    alg_status = INIT_LIB1_V2;
}

if (INIT_LIB2 >= alg_status) {
    if (MPI_SUCCESS != (ret =
       lib2_recovery(result, size_vector, *world,
       &lib2_status, 0))) {
        /* Failure during the library, perform
           recovery */
        OMPI_Comm_revoke(*world);
        repair(vector1, vector2, result,
           size_vector, *world, world);
    }
} else {
    if (MPI_SUCCESS != (ret = lib2_init(*world,
       &lib2_status))) {
        /* Failure during the library, perform
           recovery */
```

```
            OMPI_Comm_revoke(*world);
            repair(vector1, vector2, result,
              size_vector, *world, world);
         }
         alg_status = INIT_LIB2;
      }
   } else {
      /* Set the errhandler for MCW so it gets
        propagated to all other communicators */
      MPI_Comm_set_errhandler(MPI_COMM_WORLD,
        MPI_ERRORS_RETURN);

      if (MPI_SUCCESS != (ret =
        MPI_Comm_dup(MPI_COMM_WORLD, world))) {
         /* Perform recovery */
         if (MPI_ERR_PROC_FAILED == ret) {
            OMPI_Comm_revoke(MPI_COMM_WORLD);
            repair(vector1, vector2, result,
              size_vector, MPI_COMM_WORLD, world);
         } else if (MPI_ERR_REVOKED == ret) {
            repair(vector1, vector2, result,
              size_vector, MPI_COMM_WORLD, world);
         }
      }

      if (LIB1_SUCCESS != (ret = lib1_init(*world,
        &lib1_status1))) {
```

```
    /* Failure during the library, perform
       recovery */
    OMPI_Comm_revoke(*world);
    repair(vector1, vector2, result, size_vector,
       *world, world);
}

alg_status = INIT_LIB1_V1;

if (LIB1_SUCCESS != (ret = lib1_init(*world,
   &lib1_status2))) {
    /* Failure during the library, perform
       recovery */
    OMPI_Comm_revoke(*world);
    repair(vector1, vector2, result, size_vector,
       *world, world);
}

alg_status = INIT_LIB1_V2;

if (LIB2_SUCCESS != (ret = lib2_init(*world,
   &lib2_status))) {
    /* Failure during the library, perform
       recovery */
    OMPI_Comm_revoke(*world);
    repair(vector1, vector2, result, size_vector,
       *world, world);
}
```

```
    alg_status = INIT_LIB2;

    MPI_Comm_rank(*world, &rank);
    MPI_Comm_size(*world, &size);

    if (0 != size_vector % (size-1)) {
        fprintf(stderr, "Invalid job size. The size
           of the vector needs to be divisible by the
           number of processes in the job - 1 (for
           checksums).\n");
        exit(1);
    }

    /* Arbitraily divide up the vector to make sure
       everyone doesn't have the same values. This is
       only an example after all... */
    size_vector /= (size - 1);

    vector1 = &vector1[rank * size_vector];
    vector2 = &vector2[rank * size_vector];
}

fprintf(stdout, "Vectors loaded...\n");
for (i = 0; i < size_vector; i++) {
    fprintf(stdout, "[%d] %f\t%f\n", rank,
       vector1[i], vector2[i]);
}
```

```
    sleep(1);
#endif

    if (SCALE1 >= alg_status) {
        if (LIB1_SUCCESS != (ret =
           lib1_min_scale_vector(vector1, size_vector,
           1000, &lib1_status1))) {
            /* Failure during the library, perform
               recovery */
            repair_comm = (MPI_Comm *)
               malloc(sizeof(MPI_Comm));

            done = 0;

            /* Revoke and repair the old communicator */
            OMPI_Comm_revoke(*world);
            repair(vector1, vector2, result, size_vector,
               *world, repair_comm);
            MPI_Comm_free(world);
            free(world);
            world = repair_comm;
        }

        alg_status = SCALE1;
    }

    if (SCALE2 >= alg_status) {
```

```
    if (LIB1_SUCCESS != (ret =
       lib1_min_scale_vector(vector2 , size_vector ,
       1000 , &lib1_status2))) {
        /* Failure during the library , perform
           recovery */
        repair_comm = (MPI_Comm *)
           malloc(sizeof(MPI_Comm));

        done = 0;

        /* Revoke and repair the old communicator */
        OMPI_Comm_revoke(*world);
        repair(vector1 , vector2 , result , size_vector ,
           *world , repair_comm);
        MPI_Comm_free(world);
        free(world);
        world = repair_comm;
    }

    alg_status = SCALE2;
}

if (ADD >= alg_status) {
    if (LIB2_SUCCESS != (ret =
       lib2_vector_add(vector1 , vector2 , size_vector ,
       result , &lib2_status))) {
        /* Failure during the library , perform
           recovery */
        repair_comm = (MPI_Comm *)
           malloc(sizeof(MPI_Comm));

        done = 0;

        /* Revoke and repair the old communicator */
        OMPI_Comm_revoke(*world);
        repair(vector1, vector2, result, size_vector,
           *world, repair_comm);
        MPI_Comm_free(world);
        free(world);
        world = repair_comm;
    }

    alg_status = ADD;
}

if (0 == rank) {
    final_total = (float *) malloc(sizeof(float) *
       size_vector * size);
}
if (MPI_SUCCESS != (ret = MPI_Gather(result,
   size_vector, MPI_FLOAT, final_total, size_vector,
   MPI_FLOAT, 0, *world))) {
    /* Perform recovery */
    OMPI_Comm_revoke(*world);
    repair(vector1, vector2, result, size_vector,
       *world, repair_comm);
```

```
        MPI_Comm_free(world);
        free(world);
        world = repair_comm;
    }

    if (0 == rank) {
        fprintf(stdout, "\n---Result---\n");
        for (i = 0; i < size_vector * size; i++) {
            fprintf(stdout, "%f\n", final_total[i]);
        }
    }

    MPI_Finalize();

    return 0;
}

void repair(float *v1, float *v2, float *result, int
   size_v, MPI_Comm broken, MPI_Comm *repaired) {
    MPI_Comm temp, temp_intercomm, temp_intracomm,
       *recursive_repair;
    int ret, *errcodes, procs_needed, old_rank, i,
       new_rank, old_group_size;
    int *temp_ranks, *failed_ranks, *new_ranks;
    MPI_Group old_group, failed_group, new_group;
    enum progression best_status;
```

```
/* Get the needed data about the broken communicator
   */
MPI_Comm_size(broken, &old_group_size);
MPI_Comm_group(broken, &old_group);
MPI_Comm_rank(broken, &old_rank);
OMPI_Comm_failure_ack(broken);
OMPI_Comm_failure_get_acked(broken, &failed_group);
MPI_Group_size(failed_group, &procs_needed);
errcodes = (int *) malloc(sizeof(int) * procs_needed);

/* Figure out ranks of the processes which had failed
   */
temp_ranks = (int *) malloc(sizeof(int) *
   old_group_size);
failed_ranks = (int *) malloc(sizeof(int) *
   old_group_size);
for (i = 0; i < old_group_size; i++) {
    temp_ranks[i] = i;
}
MPI_Group_translate_ranks(failed_group, procs_needed,
   temp_ranks, old_group, failed_ranks);
MPI_Group_free(&old_group);
MPI_Group_free(&failed_group);

/* Shrink the broken communicator to remove failed
   procs */
OMPI_Comm_shrink(broken, &temp);
```

```
/* Spawn the new process(es) */
if (MPI_SUCCESS != (ret =
    MPI_Comm_spawn("./vector_math ", NULL,
       procs_needed, MPI_INFO_NULL, 0, temp,
       &temp_intercomm, errcodes))) {
    free(temp_ranks);
    free(failed_ranks);
    free(errcodes);
    MPI_Comm_free(&temp);
    if (MPI_ERR_PROC_FAILED == ret) {
        OMPI_Comm_revoke(temp);
        return repair(v1, v2, result, size_v, broken,
           repaired);
    } else if (MPI_ERR_REVOKED == ret) {
        return repair(v1, v2, result, size_v, broken,
           repaired);
    } else {
        fprintf(stderr, "Unknown error with
           MPI_COMM_SPAWN: %d\n", ret);
        exit(1);
    }
}
free(errcodes);
MPI_Comm_free(&temp);

/* Merge the new processes into a new communicator */
```

```
if (MPI_SUCCESS != (ret =
  MPI_Intercomm_merge(temp_intercomm, 0,
  &temp_intracomm))) {
   free(temp_ranks);
   free(failed_ranks);
   MPI_Comm_free(&temp_intercomm);
   if (MPI_ERR_PROC_FAILED == ret) {
      /* Start the recovery over again if there is
        a failure. */
      OMPI_Comm_revoke(temp_intercomm);
      return repair(v1, v2, result, size_v, broken,
        repaired);
   } else if (MPI_ERR_REVOKED == ret) {
      /* Start the recovery over again if there is
        a failure. */
      OMPI_Comm_revoke(temp_intercomm);
      return repair(v1, v2, result, size_v, broken,
        repaired);
   } else {
      fprintf(stderr, "Unknown error with
        MPI_COMM_SPAWN: %d\n", ret);
      exit(1);
   }
}
MPI_Comm_free(&temp_intercomm);

/* Tell the new processes what their old ranks were */
MPI_Comm_rank(temp_intracomm, &new_rank);
```

```
if (0 == new_rank) {
    MPI_Comm_group(temp_intracomm, &new_group);
    new_ranks = (int *) malloc(sizeof(int) *
       procs_needed);
    MPI_Group_translate_ranks(new_group,
       procs_needed, temp_ranks, new_group,
       new_ranks);
    MPI_Group_free(&new_group);

    for (i = 0; i < procs_needed; i++) {
        if (MPI_SUCCESS != (ret =
            MPI_Send(&failed_ranks[i], 1, MPI_INT,
               new_ranks[i], 31337, temp_intracomm)))
               {
            free(temp_ranks);
            free(failed_ranks);
            free(new_ranks);
            if (MPI_ERR_PROC_FAILED == ret) {
                /* Start the recovery over again if
                   there is a failure. */
                OMPI_Comm_revoke(temp_intercomm);
                return repair(v1, v2, result, size_v,
                   broken, repaired);
            } else if (MPI_ERR_REVOKED == ret) {
                /* Start the recovery over again if
                   there is a failure. */
                OMPI_Comm_revoke(temp_intercomm);
                return repair(v1, v2, result, size_v,
                   broken, repaired);
            } else {
                fprintf(stderr, "Unknown error with
                   MPI_SEND: %d\n", ret);
                exit(1);
            }
        }
    }
}
free(temp_ranks);
free(failed_ranks);
free(new_ranks);

/* Everyone move to their old position in the
   recovered communicator */
if (MPI_SUCCESS != (ret =
   MPI_Comm_split(temp_intracomm, 0, old_rank,
   repaired))) {
    if (MPI_ERR_PROC_FAILED == ret) {
        /* Start the recovery over again if there is
           a failure. */
        OMPI_Comm_revoke(temp_intercomm);
        return repair(v1, v2, result, size_v, broken,
           repaired);
    } else if (MPI_ERR_REVOKED == ret) {
        /* Start the recovery over again if there is
           a failure. */
        OMPI_Comm_revoke(temp_intercomm);
        return repair(v1, v2, result, size_v, broken,
           repaired);
    } else {
        fprintf(stderr, "Unknown error with
           MPI_COMM_SPLIT: %d\n", ret);
        exit(1);
    }
}

/* If someone has reached this point, we should
   recover lib1 */
if (INIT_LIB1_V1 >= best_status) {
    if (LIB1_SUCCESS != lib1_recovery(v1, size_v,
       *repaired, &lib1_status1, (INIT_LIB1_V1 >=
       alg_status))) {
        recursive_repair = (MPI_Comm *)
           malloc(sizeof(MPI_Comm));
        OMPI_Comm_revoke(*repaired);
        return repair(v1, v2, result, size_v,
           *repaired, repaired);
    }
}

/* If someone has reached this point, we should
   recover lib1 */
if (INIT_LIB1_V2 >= best_status) {
        if (LIB1_SUCCESS != lib1_recovery(v1, size_v,
           *repaired, &lib1_status2, (INIT_LIB1_V2 >=
           alg_status))) {
            recursive_repair = (MPI_Comm *)
               malloc(sizeof(MPI_Comm));
            OMPI_Comm_revoke(*repaired);
            return repair(v1, v2, result, size_v,
               *repaired, repaired);
        }
    }

    /* If someone has reached this point, we should
       recover lib1 */
    if (INIT_LIB2 >= best_status) {
        if (LIB2_SUCCESS != lib2_recovery(result, size_v,
           *repaired, &lib2_status, (INIT_LIB2 >=
           alg_status))) {
            recursive_repair = (MPI_Comm *)
               malloc(sizeof(MPI_Comm));
            OMPI_Comm_revoke(*repaired);
            return repair(v1, v2, result, size_v,
               *repaired, repaired);
        }
    }

    alg_status = best_status;
}
```

B.2 Library 1

This section is the header and main code for the first library. This library performs the scaling operation. Note how the library tracks recovery status by using a status object which is actually managed by the calling code. This facilitates recover across instances in cases where a node may be migrated and the data recovered in a new location.

lib1.h

```
#ifndef LIB1_H
#define LIB1_H

#include "mpi.h"

struct lib1_status {
    /* A flag to keep track of whether or not we should
       be recovering */
    int recovering;
    /* How many iterations are left in the operation */
    int iterations_left;
    /* The checksum values (only used on the checksum
       rank) */
    float *checksum;
    MPI_Comm lib1_comm_full;
    MPI_Comm lib1_comm;
    int checksum_rank;
};
typedef struct lib1_status lib1_status_t;
```

```
int lib1_init(MPI_Comm comm, lib1_status_t *status);

int lib1_recovery(float *v, int size_v, MPI_Comm comm,
   lib1_status_t *status, int correct);

int lib1_min_scale_vector(float *v, int size_v, int
   iterations, lib1_status_t *status);

int lib1_finalize(lib1_status_t *status);

#define LIB1_SUCCESS 0
#define LIB1_FAILURE 1
#define LIB1_UNRECOVERABLE 2

#endif
```

lib1.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "mpi.h"
#include "mpi-ext.h"
#include "lib1.h"

int lib1_init(MPI_Comm comm, lib1_status_t *status) {
    int rank, size;

    status->recovering = 0;
```

```
/* Duplicate the communicator to have seperation of
  failures between libraries */
if (MPI_SUCCESS != MPI_Comm_dup(comm,
  &status->lib1_comm_full)) {
   /* Revoke the new communicator in case it was
     created somehow and return. We'll try again
     from scratch */
   OMPI_Comm_revoke(status->lib1_comm_full);

   return LIB1_UNRECOVERABLE;
}



MPI_Comm_rank(status->lib1_comm_full, &rank);
MPI_Comm_size(status->lib1_comm_full, &size);

status->checksum_rank = size - 1;

if (status->checksum_rank == rank) {
   /* Duplicate the communicator to have seperation
     of failures between libraries */
   if (MPI_SUCCESS !=
     MPI_Comm_split(status->lib1_comm_full,
     MPI_UNDEFINED, rank, &status->lib1_comm)) {
      /* Revoke the new communicator in case it was
        created somehow and return. We'll try
        again from scratch */
```

```
            OMPI_Comm_revoke(status->lib1_comm);
            OMPI_Comm_revoke(status->lib1_comm_full);

            return LIB1_UNRECOVERABLE;
        }
    } else {
        /* Duplicate the communicator to have seperation
           of failures between libraries */
        if (MPI_SUCCESS !=
           MPI_Comm_split(status->lib1_comm_full, 0,
           rank, &status->lib1_comm)) {
            /* Revoke the new communicator in case it was
               created somehow and return. We'll try
               again from scratch */
            OMPI_Comm_revoke(status->lib1_comm);
            OMPI_Comm_revoke(status->lib1_comm_full);

            return LIB1_UNRECOVERABLE;
        }
    }

    return LIB1_SUCCESS;
}

int lib1_recovery(float *v, int size_v, MPI_Comm comm,
   lib1_status_t *status, int correct) {
    int ret, size, rank, i;
    float *checksums;
```

```
/* Duplicate the communicator to have seperation of
   failures between libraries */
if (MPI_SUCCESS != MPI_Comm_dup(comm,
   &status->lib1_comm_full)) {
    /* Revoke the new communicator in case it was
       created somehow and return. We'll try again
       from scratch */
    OMPI_Comm_revoke(status->lib1_comm);

    return LIB1_FAILURE;
}

MPI_Comm_size(status->lib1_comm_full, &size);
MPI_Comm_rank(status->lib1_comm_full, &rank);

status->checksum_rank = size - 1;

if (status->checksum_rank == rank) {
    /* Duplicate the communicator to have seperation
       of failures between libraries */
    if (MPI_SUCCESS !=
       MPI_Comm_split(status->lib1_comm_full,
       MPI_UNDEFINED, rank, &status->lib1_comm)) {
        /* Revoke the new communicator in case it was
           created somehow and return. We'll try
           again from scratch */
        OMPI_Comm_revoke(status->lib1_comm);
        OMPI_Comm_revoke(status->lib1_comm_full);

        return LIB1_FAILURE;
    }
} else {
    /* Duplicate the communicator to have seperation
       of failures between libraries */
    if (MPI_SUCCESS !=
       MPI_Comm_split(status->lib1_comm_full, 0,
       rank, &status->lib1_comm)) {
        /* Revoke the new communicator in case it was
           created somehow and return. We'll try
           again from scratch */
        OMPI_Comm_revoke(status->lib1_comm);
        OMPI_Comm_revoke(status->lib1_comm_full);

        return LIB1_FAILURE;
    }
}

/* Determine whether there is anything to recover and
   inform the new process */
if (MPI_SUCCESS != MPI_Bcast(&status->recovering, 1,
   MPI_INT, status->checksum_rank,
   status->lib1_comm_full)) {
    OMPI_Comm_revoke(status->lib1_comm_full);
    OMPI_Comm_revoke(status->lib1_comm);
    return LIB1_FAILURE;
}

/* Broadcast the ABFT checksum and iterations
   remaining and use them to recover */
if (status->recovering) {
    if (status->checksum_rank != rank) {
        status->checksum = (float *)
           malloc(sizeof(float) * size_v);
    }

    if (MPI_SUCCESS != MPI_Bcast(&status->checksum,
       size_v, MPI_FLOAT, status->checksum_rank,
       status->lib1_comm_full)) {
        OMPI_Comm_revoke(status->lib1_comm_full);
        OMPI_Comm_revoke(status->lib1_comm);

        return LIB1_FAILURE;
    }

    if (rank != status->checksum_rank) {
        checksums = (float *) malloc(sizeof(float) *
           size_v);

        if (MPI_SUCCESS != (ret = MPI_Allreduce(v,
           checksums, size_v, MPI_FLOAT, MPI_SUM,
           status->lib1_comm))) {
            OMPI_Comm_revoke(status->lib1_comm_full);
            OMPI_Comm_revoke(status ->lib1_comm);

            free(checksums);
            return LIB1_FAILURE;
        }

        if (!correct) {
            for (i = 0; i < size_v; i++) {
                v[i] = status ->checksum[i] -
                   checksums[i];
            }
        }

        free(checksums);
    }

    if (MPI_SUCCESS !=
       MPI_Bcast(&status ->iterations_left , 1,
       MPI_INT , status ->checksum_rank ,
       status ->lib1_comm_full)) {
        OMPI_Comm_revoke(status ->lib1_comm_full);
        OMPI_Comm_revoke(status ->lib1_comm);

        return LIB1_FAILURE;
    }
}

return LIB1_SUCCESS;
```

```
}

int lib1_min_scale_vector(float *v,
                          int size_v,
                          int iterations,
                          lib1_status_t *status) {
    float local_min, global_min;
    int i, ret, rank, size;

    MPI_Comm_rank(status->lib1_comm_full, &rank);
    MPI_Comm_size(status->lib1_comm_full, &size);

    if (!status->recovering) {
        status->iterations_left = iterations;

        if (status->checksum_rank == rank) {
            status->checksum = (float *)
               malloc(sizeof(float) * size_v);
        }

        /* Calculate the initial checksum */
        if (MPI_SUCCESS != (ret = MPI_Reduce(v,
           status->checksum, size_v, MPI_FLOAT, MPI_SUM,
           status->checksum_rank,
           status->lib1_comm_full))) {
            OMPI_Comm_revoke(status->lib1_comm);
            OMPI_Comm_revoke(status->lib1_comm_full);
            /* We can't recover from this error as we
               haven't created the checksum yet */
            return LIB1_UNRECOVERABLE;
        }

    } else {
        status->recovering = 0;

        if (0 == status->iterations_left) {
            return LIB1_SUCCESS;
        }
    }

    for (;status->iterations_left > 0;
       status->iterations_left--) {
        if (status->checksum_rank == rank) {
            /* Calculate the min among the local values */
            local_min = v[0];
            for(i = 1; i < size_v; i++) {
                if (local_min > v[i]) {
                    local_min = v[i];
                }
            }

            /* Calculate the min among all processes */
            if (MPI_SUCCESS != (ret =
               MPI_Allreduce(&local_min, &global_min, 1,
               MPI_FLOAT, MPI_MIN, status->lib1_comm))) {
```

```
    /* Perform recovery */
    status->recovering = 1;

    /* Revoke the internal communicator and
       return. */
    OMPI_Comm_revoke(status->lib1_comm);
    OMPI_Comm_revoke(status->lib1_comm_full);

    return LIB1_FAILURE;
}

/* Update the checksum */
if (0 == rank) {
    if (MPI_SUCCESS != (ret =
       MPI_Send(&global_min, 1, MPI_FLOAT,
       status->checksum_rank, 0,
       status->lib1_comm_full))) {
        status->recovering = 1;
        OMPI_Comm_revoke(status->lib1_comm);
        OMPI_Comm_revoke(status->lib1_comm_full);

        return LIB1_FAILURE;
    }
}

/* Scale the local vector */
for (i = 0; i < size_v; i++) {
    v[i] *= global_min;
```

```
            }
        } else {
            if (MPI_SUCCESS != (ret =
              MPI_Recv(&global_min, 1, MPI_FLOAT, 0, 0,
              status->lib1_comm_full,
              MPI_STATUS_IGNORE))) {
                status->recovering = 1;

                OMPI_Comm_revoke(status->lib1_comm);
                OMPI_Comm_revoke(status->lib1_comm_full);

                return LIB1_FAILURE;
            }

            for (i = 0; i < size_v; i++) {
                status->checksum[i] *= global_min;
            }
        }
    }

    return LIB1_SUCCESS;
}

int lib1_finalize(lib1_status_t *status) {
    int done = 1, ret;

    /* Make sure everyone agrees that the operations were
       successful */
    if (MPI_SUCCESS != (ret =
       OMPI_Comm_agree(status->lib1_comm_full, &done))) {
        /* Fail out of this function, the recovering
           process will still need us to call the
           recovery function to send it the resulting
           checksum */
        status->recovering = 1;

        OMPI_Comm_revoke(status->lib1_comm);
        OMPI_Comm_revoke(status->lib1_comm_full);

        return LIB1_FAILURE;
    }

    free(status);

    return LIB1_SUCCESS;
}
```

B.3 Library 2

This section is the header and main code for the second library. This library performs the addition operation. Again, note how the library tracks recovery status by using a status object which is actually managed by the calling code. This facilitates recover across instances in cases where a node may be migrated and the data recovered in a new location.

lib2.h

```
#ifndef LIB2_H
```

```
#define LIB2_H

#include "mpi.h"

struct lib2_status {
    int recovering;
    int operation_done;
    float *checksum;
    MPI_Comm lib2_comm;
    MPI_Comm lib2_comm_full;
    int checksum_rank;
};
typedef struct lib2_status lib2_status_t;

int lib2_init(MPI_Comm comm, lib2_status_t *status);

int lib2_recovery(float *result,
                  int size_v,
                  MPI_Comm comm,
                  lib2_status_t *status,
                  int correct);

int lib2_vector_add(float *v1,
                    float *v2,
                    int size_v,
                    float *result,
                    lib2_status_t *status);
```

```
int lib2_finalize(lib2_status_t *status);

#define LIB2_SUCCESS 0
#define LIB2_FAILURE 1
#define LIB2_UNRECOVERABLE 2

#endif
```

lib2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "mpi.h"
#include "mpi-ext.h"
#include "lib2.h"

int lib2_init(MPI_Comm comm, lib2_status_t *status) {
    int rank, size;

    status->recovering = 0;

    /* Duplicate the communicator to have seperation of
       failures between libraries */
    if (MPI_SUCCESS != MPI_Comm_dup(comm,
       &status->lib2_comm_full)) {
        /* Revoke the new communicator in case it was
           created somehow and return. We’ll try again
           from scratch */
        OMPI_Comm_revoke(status->lib2_comm_full);

        return LIB2_UNRECOVERABLE;
    }

    MPI_Comm_rank(status->lib2_comm_full, &rank);
    MPI_Comm_size(status->lib2_comm_full, &size);

    status->checksum_rank = size - 1;

    if (status->checksum_rank == rank) {
        /* Duplicate the communicator to have seperation
           of failures between libraries */
        if (MPI_SUCCESS !=
           MPI_Comm_split(status->lib2_comm_full,
           MPI_UNDEFINED, rank, &status->lib2_comm)) {
            /* Revoke all communicators and start from
               scratch */
            OMPI_Comm_revoke(status->lib2_comm);
            OMPI_Comm_revoke(status->lib2_comm_full);

            return LIB2_UNRECOVERABLE;
        }
    } else {
        if (MPI_SUCCESS !=
           MPI_Comm_split(status->lib2_comm_full, 0,
           rank, &status->lib2_comm)) {
            OMPI_Comm_revoke(status->lib2_comm);
```

```
            OMPI_Comm_revoke(status->lib2_comm_full);

            return LIB2_UNRECOVERABLE;
        }
    }

    return LIB2_SUCCESS;
}

int lib2_recovery(float *result, int size_v, MPI_Comm
   comm, lib2_status_t *status, int correct) {
    int size, i, rank;
    float *checksums;

    /* Duplicate the communicator to have seperation of
       failures between libraries */
    if (MPI_SUCCESS != MPI_Comm_dup(comm,
       &status->lib2_comm_full)) {
        /* Revoke the new communicator in case it was
           created somehow and return. We'll try again
           from scratch */
        OMPI_Comm_revoke(status->lib2_comm_full);

        return LIB2_FAILURE;
    }

    MPI_Comm_size(status->lib2_comm_full, &size);
    MPI_Comm_rank(status->lib2_comm_full, &rank);
```

```
status->checksum_rank = size -1;

if (status->checksum_rank == rank) {
    /* Duplicate the communicator to have seperation
       of failures between libraries */
    if (MPI_SUCCESS !=
       MPI_Comm_split(status->lib2_comm_full,
       MPI_UNDEFINED, rank, &status->lib2_comm)) {
        OMPI_Comm_revoke(status->lib2_comm);
        OMPI_Comm_revoke(status->lib2_comm_full);

        return LIB2_FAILURE;
    }
} else {
    /* Duplicate the communicator to have seperation
       of failures between libraries */
    if (MPI_SUCCESS !=
       MPI_Comm_split(status->lib2_comm_full, 0,
       rank, &status->lib2_comm)) {
        OMPI_Comm_revoke(status->lib2_comm);
        OMPI_Comm_revoke(status->lib2_comm_full);

        return LIB2_FAILURE;
    }
}
```

```
/* Determine whether there is anything to recover and
   inform the new process */
if (MPI_SUCCESS != MPI_Bcast(&status->recovering, 1,
   MPI_INT, status->checksum_rank,
   status->lib2_comm_full)) {
    OMPI_Comm_revoke(status->lib2_comm_full);
    OMPI_Comm_revoke(status->lib2_comm);

    return LIB2_FAILURE;
}

/* Broadcast the ABFT checksum and whether or not we
   were done with the
 * operation */
if (status->recovering) {
    if (status->checksum_rank != rank) {
        status->checksum = (float *)
           malloc(sizeof(float) * size_v);
    }

    if (MPI_SUCCESS != MPI_Bcast(&status->checksum,
       size_v, MPI_FLOAT, status->checksum_rank,
       status->lib2_comm)) {
        OMPI_Comm_revoke(status->lib2_comm_full);
        OMPI_Comm_revoke(status->lib2_comm);

        return LIB2_FAILURE;
    }
```

```
if (rank != status->checksum_rank) {
    checksums = (float *) malloc(sizeof(float) *
       size_v);

    if (MPI_SUCCESS != MPI_Allreduce(result,
       checksums, size_v, MPI_FLOAT, MPI_SUM,
       status->lib2_comm)) {
        OMPI_Comm_revoke(status->lib2_comm_full);
        OMPI_Comm_revoke(status->lib2_comm);
        free(checksums);

        return LIB2_FAILURE;
    }

    if (!correct) {
        for (i = 0; i < size_v; i++) {
            result[i] = status->checksum[i] -
               checksums[i];
        }
    }

    free(checksums);
}
```

```
        if (MPI_SUCCESS !=
           MPI_Bcast(&status->operation_done, 1, MPI_INT,
           status->checksum_rank,
           status->lib2_comm_full)) {
            OMPI_Comm_revoke(status->lib2_comm_full);
            OMPI_Comm_revoke(status->lib2_comm);

            return LIB2_FAILURE;
        }
    }

    return LIB2_SUCCESS;
}

int lib2_vector_add(float *v1,
                    float *v2,
                    int size_v,
                    float *result,
                    lib2_status_t *status) {
    int i, rank, size;
    float *temp;
    MPI_Status mpi_status;

    MPI_Comm_rank(status->lib2_comm_full, &rank);
    MPI_Comm_size(status->lib2_comm_full, &size);

    if (!status->recovering) {
        status->operation_done = 0;
        if (status->checksum_rank == rank) {
            status->checksum = (float *) calloc(size_v,
               sizeof(float));
        }
    } else {
        status->recovering = 0;

        if (status->operation_done) {
            return LIB2_SUCCESS;
        }
    }

    temp = (float *) malloc(sizeof(float) * size_v);

    if (status->checksum_rank == rank) {
        if (MPI_SUCCESS != MPI_Sendrecv(v1, size_v,
           MPI_FLOAT, (size-1-rank), 31337, temp, size_v,
           MPI_FLOAT, (size-1-rank), 31337,
           status->lib2_comm, &mpi_status)) {
            /* Perform recovery */
            status->recovering = 1;

            OMPI_Comm_revoke(status->lib2_comm);

            return LIB2_FAILURE;
        }
```

```
        for (i = 0; i < size_v; i++) {
            result[i] = temp[i] + v2[i];
        }
    }

    /* Update the checksum */
    if (MPI_SUCCESS != MPI_Reduce(result,
       status->checksum, size_v, MPI_FLOAT, MPI_SUM,
       status->checksum_rank, status->lib2_comm_full)) {
        status->recovering = 1;

        OMPI_Comm_revoke(status->lib2_comm);
        OMPI_Comm_revoke(status->lib2_comm_full);

        return LIB2_UNRECOVERABLE;
    }

    status->operation_done = 1;

    return LIB2_SUCCESS;
}

int lib2_finalize(lib2_status_t *status) {
    int ret, done = 1;

    /* Make sure everyone agrees that the operations were
       successful */
    if (MPI_SUCCESS != (ret =
       OMPI_Comm_agree(status->lib2_comm_full, &done))) {
        /* Fail out of this function, the recovering
           process will still need us to call the
           recovery function to send it the resulting
           checksum */
        status->recovering = 1;

        OMPI_Comm_revoke(status->lib2_comm);
        OMPI_Comm_revoke(status->lib2_comm_full);

        return LIB2_FAILURE;
    }

    free(status);

    return LIB2_SUCCESS;
}
```

Vita

Wesley Bland was born in Knoxville, TN on April 5, 1985. He graduated from Farragut High School in May of 2003 and began undergraduate studies at Tennessee Technological University in Cookeville, TN. In 2003 he finished a Bachelor's degree in Computer Science and moved back to Knoxville to continue his studies at the University of Tennessee. During this time, he worked at Oak Ridge National Laboratory as an intern, primarily in the cluster computing group led by Stephen L. Scott and the application performance tools group led by Richard Graham. In 2009, Wesley finished his Master's Degree in Computer Science under Jack Dongarra in the Innovative Computing Laboratory. He continued to work with George Bosilca's team in the ICL on projects related to MPI and Fault Tolerance until completing his Doctor of Philosophy degree in Computer Science in May 2013.