LAZY FAULT RECOVERY FOR REDUNDANT MPI

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Elie Saliba

June 2019
COMMITTEE MEMBERSHIP

TITLE: Lazy Fault Recovery for Redundant MPI

AUTHOR: Elie Saliba

DATE SUBMITTED: June 2019

COMMITTEE CHAIR: Maria Pantoja, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Christopher Lupo, Ph.D.
Professor of Computer Science

COMMITTEE MEMBER: Hugh Smith, Ph.D.
Professor of Computer Science
ABSTRACT

Lazy Fault Recovery for Redundant MPI

Elie Saliba

Distributed Systems (DS) where multiple computers share a workload across a network, are used everywhere, from data intensive computations to storage and machine learning. DS provide a relatively cheap and efficient solution that allows stability with improved performance for computational intensive applications. In a DS faults and failures are the norm not the exception. At any moment data corruption can occur especially since a DS usually consists of hundred to thousands of units of commodity hardware. The large number and quality of components guarantees, by probability, that at any given time some of the components will not be working and some of them will not recover from failure. DS can experience problems caused by application bugs, operating systems bugs, failures with disks, memory, connectors, networking, power supply, and other components; therefore, constant monitoring and failure detection are fundamental. Automatic recovery must be integral to the system. One of the most commonly used programming languages for DS is Message Passing Interface (MPI). Unfortunately MPI does not support fault detection or recovery. In this thesis, we build a recovery mechanism based on replicas that works on top of the asynchronous fault detection implemented in previous work. Results shows that our recovery implementation is successful and the overhead in execution time is minimal.
ACKNOWLEDGMENTS

Thanks to:

• My parents for funding my education and believing in me.

• My family and friends for the continuous support in my journey.

• My advisor Dr. Pantoja for guiding me through this work.

• My IG followers.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>1  Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2  Background</td>
<td>3</td>
</tr>
<tr>
<td>2.1 MPI</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Origins and Features</td>
<td>3</td>
</tr>
<tr>
<td>2.1.2 Goals</td>
<td>4</td>
</tr>
<tr>
<td>2.1.3 Basic Concepts</td>
<td>5</td>
</tr>
<tr>
<td>2.1.4 Versions</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Resilience</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Checkpoint/Rerstart</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Replication/Redundancy</td>
<td>12</td>
</tr>
<tr>
<td>2.5 RedMPI</td>
<td>14</td>
</tr>
<tr>
<td>2.5.1 All-to-All</td>
<td>15</td>
</tr>
<tr>
<td>2.5.2 Message Plus Hash (MsgPlusHash)</td>
<td>17</td>
</tr>
<tr>
<td>2.5.3 Async</td>
<td>18</td>
</tr>
<tr>
<td>2.5.4 AsyncHash</td>
<td>20</td>
</tr>
<tr>
<td>3  Implementation</td>
<td>21</td>
</tr>
<tr>
<td>3.1 Async Recovery</td>
<td>21</td>
</tr>
<tr>
<td>3.1.1 Edge Cases</td>
<td>22</td>
</tr>
<tr>
<td>3.2 Fault Injector</td>
<td>24</td>
</tr>
</tbody>
</table>


LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Execution time in seconds for NPB using the MsgPlusHash and AsyncHash functions</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>Execution time in seconds for NPB using the AllToAll and Async functions</td>
<td>29</td>
</tr>
<tr>
<td>4.3</td>
<td>Execution time in seconds for the Addition benchmark using Async and Async recovery</td>
<td>38</td>
</tr>
<tr>
<td>4.4</td>
<td>Execution time in seconds for the matrix multiplication benchmark using Async and Async recovery</td>
<td>39</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>MPI basic six functions [19]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>100K Node Job with Varied MTBF [21]</td>
<td>11</td>
</tr>
<tr>
<td>2.3</td>
<td>Modeled Time to Completion with Redundancy [23]</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Redundancy Representation [23]</td>
<td>15</td>
</tr>
<tr>
<td>2.5</td>
<td>All-to-All Method Overview [23]</td>
<td>16</td>
</tr>
<tr>
<td>2.6</td>
<td>MsgPlusHash Method Overview [23]</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>Async Method Overview [24]</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Runtime measurements for SP(B) benchmark using MsgPlusHash and AsyncHash</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>Runtime measurements for SP(B) benchmark using AllToAll and Async</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Runtime measurements for BT(B) benchmark using MsgPlusHash and AsyncHash</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Runtime measurements for BT(B) benchmark using AllToAll and Async</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>Runtime measurements for LU(B) benchmark using MsgPlusHash and AsyncHash</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Runtime measurements for LU(B) benchmark using AllToAll and Async</td>
<td>34</td>
</tr>
<tr>
<td>4.7</td>
<td>Runtime measurements for CG(C) benchmark using MsgPlusHash and AsyncHash</td>
<td>35</td>
</tr>
<tr>
<td>4.8</td>
<td>Runtime measurements for CG(C) benchmark using AllToAll and Async</td>
<td>35</td>
</tr>
<tr>
<td>4.9</td>
<td>Async correction</td>
<td>37</td>
</tr>
<tr>
<td>4.10</td>
<td>First replica exiting</td>
<td>38</td>
</tr>
</tbody>
</table>
4.11 Second and third Replica exiting ........................................ 38
4.12 Runtime measurements for the addition benchmark using Async and Async Recovery ........................................ 38
4.13 Runtime measurements for the matrix multiplication benchmark using Async and Async Recovery ........................................ 40
6.1 Execution Timeline of RMT: (a) Adaptive Redundant Multithreading (b) aRMT with Lazy Fault Detection [33] ....................... 45
Chapter 1

INTRODUCTION

Message Passing Interface (MPI) is a communication standard used to define how message passing libraries should function. It primarily addresses the message-passing parallel programming model where data is moved from the address space of one process to another through cooperative means. It is mainly used in distributed systems to allow communication between nodes in a high performance computing (HPC) environment. The first official release of MPI was in June 1994 after several years of design and development. However, the work on it began in 1991 [54].

MPI inherently does not support fault detection therefore when an application is running, silent faults can occur which leads to failures. Such faults are known as silent data corruption (SDC). Mean time between failures (MTBF) is an important measure that reflects the importance of fault detection. HPC systems typically run computations that could take up to days to be performed and with MTBF being in the order of an hour or less, detection of SDCs becomes more important. This concept also introduces the necessity of having optimized SDC detection algorithms that will execute in times smaller than MTBF. Different libraries and schemes have been developed to solve this problem.

Multiple libraries introduced in the last ten years tried to address the MPI resilience problem. MR-MPI developed at Oak Ridge National Laboratory and rMPI at Sandia National Laboratories [20] were first on the list of resilient MPI libraries. Both prototypes implement replication as a solution to SDC detection. A redundantly executed application that uses MPI will allow such libraries to intercept communications calls.
in order to compare values between replicas. The main difference between MR-MPI and rMPI is that MR-MPI does not rely on any specific MPI library whereas rMPI is specific to MPICH. VolpexMPI is another redundant MPI application that was built from scratch at the University of Houston, and does not support all functionalities of the MPI library [18].

RedMPI was developed by researchers at Oak Ridge National Laboratory in 2012. It is a library that implements replication with SDC detection by comparing messages sent and received via MPI communication. RedMPI is based on two previous prototypes, rMPI and MR-MPI. Besides detection, RedMPI supports recovery in case of using a triple redundancy. The library's functions work in a synchronous fashion [38]. In 2016 extensions to RedMPI were added which allowed asynchronous SDC detection but did not support recovery. This new method of detection is known as lazy fault detection since it does its work asynchronously which allows more flexibility in replica process execution [25, 24]. The extensions added good improvement on performance since they decreased the wait time before replicas could be compared. Therefore, added recovery to those extensions would be a big improvement to the current RedMPI. In this work, we present the addition of fault recovery to the existing async fault detection previously developed. We test our recovery success using built-in fault injectors in RedMPI. The performance of the added functionality is tested across multiple benchmarks.

The remainder of this thesis is organized as follows. Chapter 2 introduces the MPI and RedMPI libraries as well as resilience and existing solutions to that problem. Chapter 3 discusses the design and implementation of our work based on the previous implemented detection. In chapter 4 the performance is tested we then discuss some aspects of this library in chapter 5. Chapter 6 mentions related work in this field, and finally chapter 7 concludes and presents future work possibilities.
MPI resilience became an important topic over the recent years and multiple solutions were presented so far. Checkpoint/Restart and replication are the most common solutions for this problems. We begin by giving a general background on MPI, then explain the need for resilience in this library, and finally we present current solutions as well as a detailed explanation of the library we are building on top of.

2.1 MPI

The Message Passaging Interface (MPI) is a standardized library designed and developed by industry and academia researchers. It is an API that defines functions to simplify message-passing programs especially in distributed systems. The library is used for applications written in C and Fortran 77 with additional languages bindings such as Java and Python.

2.1.1 Origins and Features

MPI is the work of 80 people from 40 different groups that started in 1991. The first version was published in June 1994. The purpose of this library was to simplify message passing in distributed systems. The major feature of MPI is portability as it allows the same message-passing code to be run on different machines regardless of the characteristics of each machine as long as it supports the MPI library. It is important to realize that MPI not only can be run in a distributed system but also
support multithreading processing on a single machine. This allows users to easily write code using MPI, debug it, and test it on a local machine before scaling it to a larger distributed system. Another important feature of MPI is the ability to run transparently. Users do not have to worry about the underlying architecture of the messages being sent and received. MPI is able to do the necessary conversions between data types when information is being transferred between nodes. For example, an Int variable is of size 4 bytes on a 32 bit processor machine whereas it is 8 bytes on the newer 64 bit processor machines. The conversion between the two is therefore done by MPI automatically. MPI will also use correct communication protocols when sending/receiving data over WLAN/LAN [54].

MPI has also important design goals such as efficiency in implementing, avoidance of unnecessary work, overlapping communication, and scalability. MPI can be easily implemented since the user does not have to worry about how operations will take place on different machines. MPI uses opaque objects which hides the details of how they are represented in order to simplify its usage. It avoids unnecessary work by avoiding large amounts of extra information in messages, reusing previous computations through persistent computation and cache, and avoiding extra copying and buffering. Overlapping communication calls was achieved by implementing nonblocking functions. Finally, scalability, a needed goal in any parallel environment, benefits well from MPI as it uses subgroups of processes to scale properly [54].

2.1.2 Goals

The goals of MPI as listed by [54] can be summarized as follow:

- Implement MPI as an API easy to use by programmers.
- Efficient communication provided by overlapping computation and data sharing.
• Transparent implementation capability.

• Support for C and Fortran 77 with other bindings.

• Provide a reliable, intuitive, and easy interface to utilize.

• Allow for thread-safety.

2.1.3 Basic Concepts

Basic functionality of MPI can be summarized by the send and receive functions. MPI is a message-interface thus the importance of these two calls. A minimal send operation needs the address of the data, its length, a destination, and a tag to identify a matching receiver. Similarly, receive requires an address to store the data, the length that could be received, source, a tag, and the actual length received. Therefore a basic representation of such calls is send(address, length, destination, tag) and receive(address, length, source, tag, actlen) [29].

These simplified versions of send and receive had to be modified to support more complex scenarios. First, data in memory is not contiguous. Therefore data has to be described in a distributed manner. Second, data is stored in machines differently so knowing the type of it would facilitate communication. MPI resolved these two issues by defining messages as a triple (address, count, datatype) which translates into a message starting at address and having a count occurrences of data types datatype. Communicators is another important concept that MPI provides. MPI defines a group of processes where each has a rank as identification. Therefore, a communicator simplifies the exchange of data. The updated send and receive calls become MPI_Send(address, count, datatype, destination, tag, comm) and MPI_Recv(address, maxcount, datatype, source, tag, comm, status) [29].
The overall operation of an MPI executed program in its simplest form can be summarized by the six functions shows in Figure 2.1.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INIT</td>
<td>Initiate an MPI computation.</td>
</tr>
<tr>
<td>MPI_FINALIZE</td>
<td>Terminate a computation.</td>
</tr>
<tr>
<td>MPI_COMM_SIZE</td>
<td>Determine number of processes.</td>
</tr>
<tr>
<td>MPI_COMM_RANK</td>
<td>Determine my process identifier.</td>
</tr>
<tr>
<td>MPI_SEND</td>
<td>Send a message.</td>
</tr>
<tr>
<td>MPI_RECV</td>
<td>Receive a message.</td>
</tr>
</tbody>
</table>

Figure 2.1: MPI basic six functions [19]

The definitions of the above functions implement in C are as follow [1]

- int MPI_Init (int *argc, char **argv)
  argc and argv are passed in from the main program’s arguments.

- int MPI_Comm_size (MPI_Comm comm, int *size)
  where comm is the communicator used in the program and size is the number of processes.

- int MPI_Comm_rank (MPI_Comm comm, int *rank)
  where comm is the communicator and rank is the process id in the group of comm.

- int MPI_Finalize()

- int MPI_Send (void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
where `buf` is the address of the buffer to be sent, `count` is the number of elements to be sent, `datatype` is the type of the elements being sent, `dest` is process id of the destination, `tag` is the descriptor of the message, and `comm` is the communicator used.

- `int MPI_Recv (void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)`

  where `buf` is the address of the receiving buffer, `count` is the size of the receiving buffer, `datatype` is the type of data being received, `source` is the process id of the source or could be replaced with `MPI_ANY_SOURCE` if source is unknown, `tag` is the descriptor of the message, `comm` is the communicator being used, and `status` is the status object.

2.1.4 Versions

MPI has gone through different updates throughout the years. Below is a summary of all MPI versions as it helps understand the history behind this library [2].

- **Version 1.0: May 1994** This is the first version of MPI that was introduced after the collective work that started in 1991. This first publication includes the technical features and goals of MPI.

- **Version 1.1: June 1995** Changes in this version are minor as it mostly corrected error from version 1.0.

- **Version 1.2: July 1997** Minor changes and corrections were published. The most important added feature was a function that allows users to determine which version of MPI their implementation conforms to.
• **Version 2.0:** Jul 1997 This version focuses on extensions that cover process creation, one-sided and collective communications, external interfaces, and parallel I/O.

• **Version 1.3:** May 2008 This document combines MPI 1.1 and 1.2.

• **Version 2.1:** June 2008 This document combines MPI 1.3 and 2.0.

• **Version 2.2:** September 2009 This document mostly contains correction for to the 2.1 document with a few minor extensions.

• **Version 3.0:** September 2012 This document introduced big important extensions to MPI. This includes nonblocking collective communication, new one-sided communication operations, and binding to FORTRAN 2008. This update is considered major compared to the previous updates.

• **Version 3.1:** June 2015 The latest MPI version with the largest change being corrections to the new FORTRAN bindings in addition to some minor extensions.

It is important to note that in updated versions the new features did not have to be adopted by users.

### 2.2 Resilience

The term *resilience* has been used for multiple decades in different fields. In computing, resilience often refers to fault-tolerance. A resilient system is therefore considered a reliable system where faults can be detected and recovered [53]. In order to understand the problem of resilience, a well-defined taxonomy is needed [3].
• A **service failure** also referred to as **failure** is the event of deviation from correct service. It is the transition from correct to incorrect service due to different reasons.

• An **error** is the deviation that occurs in order to cause failures. An error that has occurred but is not detected is called a **latent** error.

• A **fault** is the cause of an error. Faults can be internal or external. Internal faults are usually caused by component failures such as resistors. A fault is **active** when it causes an error, otherwise it is **dormant**.

Resilience is an important and complicated subject in HPC. Over the years, multiple measures were introduced to prevent data corruption. Error correcting code (ECC) used in DRAM and lately deployed in server-market processes such as AMD and Intel Xeon has been historically the primary defense against silent data corruption (SDC). SDC is defined as an error that changes the system’s memory and is not detected by the hardware. Hardware protection against SDCs becomes expensive and could decrease performance which introduces the importance of software in detecting SDCs as hardware redundancy is extremely expensive [23].

Studies done at Los Alamos National Laboratory (LANL) show that the number one cause of node failures in HPC is caused by soft errors [42]. Other research conducted at Lawrence Livermore National Laboratory (LLNL) point out the severe impact that SDCs have in HPC systems. When scaling up HPC systems, SDC protection is highly needed [10].

Fault tolerance in HPC can also be enhanced by the awareness of the underlying hardware characteristics. Brandt et al. [9] implemented an API that allows the system to query a list of its elements with associated characteristics in order to assign resources to them. Findings proved that awareness of the relative health of hardware
components can improve HPC tolerance.

The most common software solutions for fault-tolerance are checkpoint/restart and replication. The following concepts are detailed in the next sections.

2.3 Checkpoint/Restart

Checkpoint/Restart (C/R) has been used as a solution for HPC resilience since the early 1990s. Different types of C/R were deployed over the years. Application-level C/R has been the predominant fault tolerance methods for decades. Since 2004, the C3 [50] toolkit software offers compile-assistance to any MPI application so it becomes self-C/R. C/R at the system-level first appeared in 1995 and is mainly implemented through the Berkeley Lab Checkpoint Restart (BLCR) [31] but it is only employed at a few HPC centers. Diskless-C/R deployed in 1997 offered checkpointing to neighbor compute nodes which could scale well in distributed systems such as the Scalable C/R library (SCR) that was released in 2009 [11]. Message logging, algorithm-based fault tolerance, proactive fault tolerance and Byzantine fault tolerance have also been researched but are not available in production HPC systems [17].

The concept of C/R is simply to save the state of a thread on a stable storage and in case of failure roll back to that state. In centralized systems, C/R is straightforward as the thread rolls back to the saved state and continues execution. In a distributed system, the asynchronous interaction between nodes makes C/R more complicated [41]. Two different algorithms were presented to solve this issue; independent checkpointing and consistent checkpointing. Independent checkpointing where each process keeps track of its state independently and on failure will rollback to its saved state. However this may cause a rollback in other processes as well in case they are dependent on the failed process. In consistent checkpointing, processes
coordinate their checkpointing such as the set of all checkpoints forms a consistent global state of the system. When a process fails, all processes have to roll back to their most recent checkpoint. Multiple optimizations were made to improve consistent checkpointing such as periodic checkpointing, decreasing the number of processes involved in the C/R operations, allowing unlimited concurrency in C/R, and using a distributed virtual memory to save checkpoints [35, 40, 5, 61, 56].

C/R is a solution but is not the most efficient. When the core count increases, so does the overhead for C/R and it does so at an exponential rate. A study made at Los Alamos National Laboratory show that on a petaflop machine, a 100 hour job was estimated to increase 251 hours due to C/R overhead. Other studies have showed that efficiency of C/R can be as high as 85% and as low as 55%. Recent work by Sandia [21] shows that only 35% of the work is assigned to computation for a 128 hour job on 100K nodes with a node MTBF of 5 years while the remainder is spend on checkpointing, restarting, and then partial recomputation of the work lost since the last checkpoint [23].

<table>
<thead>
<tr>
<th>job work</th>
<th>MTBF</th>
<th>work</th>
<th>checkpoint</th>
<th>recomp.</th>
<th>restart</th>
</tr>
</thead>
<tbody>
<tr>
<td>168 hrs.</td>
<td>5 yrs</td>
<td>35%</td>
<td>20%</td>
<td>10%</td>
<td>35%</td>
</tr>
<tr>
<td>700 hrs.</td>
<td>5 yrs</td>
<td>38%</td>
<td>18%</td>
<td>9%</td>
<td>43%</td>
</tr>
<tr>
<td>5,000 hrs.</td>
<td>1 yr</td>
<td>5%</td>
<td>5%</td>
<td>5%</td>
<td>85%</td>
</tr>
</tbody>
</table>

Figure 2.2: 100K Node Job with Varied MTBF [21]

Figure 2.2 shows that for longer-running jobs, useful work becomes insignificant to the amount of time spent on checkpointing, restarting, and recomputation. Sandia researchers came to a conclusion that replication could be a potential improvement for fault detection and recovery in large-scale systems. The most important finding is
that a state machine replication approach to exascale resilience outperforms traditional checkpoint/restart [21].

2.4 Replication/Redundancy

Replication or redundancy is another well-known solution for resilience. When using redundancy the state of the machine [49] is replicated such that if the process fails, its replica could be used to pick up the work from where it stopped. Process replication can be expensive however the improvement it has on fault detection and recovery is far more appealing. Sandia Lab discusses the importance of redundancy as a primary exascale fault tolerance mechanism, with checkpoint/restart providing secondary fault tolerance when necessary [21].

Fiala [23] demonstrates the viability of replication as a solution for fault-tolerance by conducting experiments on a 128 hour job without checkpointing on Jaguar (18,688 nodes) a super computer built a Oak Ridge National Laboratory. The job was run without redundancy (1x), dual redundancy (2x) and triple redundancy (3x) at different node counts (see Figure 2.3). Results show that with no replication (1x) the job runs are about 7% faster than dual (2x) and 20% faster than triple (3x) redundancy.
The more interesting results from the graph is looking at dual redundancy. In order to ensure the absence of SDC, a user has to run the same application twice with a total of 280 hours (twice 140, line A) whereas using double the amount of nodes with a redundancy of 2x gives a total runtime of 155 hours (line C). Hence, dual redundancy, results in nearly half the wall-clock time which is valuable in case SDC detection is important for the job being run.

Moreover, to model the importance of triple redundancy that could correct errors without the need of jobs being rerun, we compare the total runtime using third of the total number of nodes vs all nodes with triple redundancy. At 1x, having to run the job three times gives a total of 390 hours (three times 130, line A) vs running the job with triple redundancy using all nodes with a runtime of 180 hours (line C).

Replication has therefore become an important aspect in fault detection and recovery. It is used in different libraries such as rMPI, mrMPI, and RedMPI [21, 22, 30].
2.5 RedMPI

RedMPI is an MPI library that is capable of both detecting and correcting SDC faults. The replicas compare received messages or hashes from multiple senders in order to detect corrupted messages. When running in double redundant mode, RedMPI can detect corruption due to the fact that replicas will be run in parallel and therefore allow for comparison on the receiving side. When running in triple redundant mode, RedMPI uses a voting algorithm to determine which of the received messages are correct and should be used by all receivers. It is important to mention that RedMPI does not protect an interconnect since it only runs in the scope of an application and its data. This also entails that RedMPI does not protect messages over a transport layer such as TCP and assumes the transport to be reliable. RedMPI does not protect against I/O functionality nor application code or instructions.

In order to understand how RedMPI functions internally, it is important to understand how internal replication works. When running a MPI job, degree of replication needs to be specified. On runtime, the total number of processes needs to be specified as well as the total replicated processes. For example, running a job with 128 processes with redundancy degree 2 would mean running a total of 256 processes. Figure 2.4 shows how triple redundancy would appear within a MPI application expecting three processes. The actual rank assigned by `mpiexec/mpirun` is referred to as native rank. The number of replicas for each virtual rank is the redundancy of the application knows as the replication degree. Within RedMPI, a mapping is stored in each process that allows the forward and reverse lookup of any processes native rank, virtual rank, or replica rank. RedMPI automatically generates a structures that maps native ranks to a virtual rank of [0...virtual_size - 1] and assign replica ranks of [0...(native_size/virtual_size)]. RedMPI allows users to input a custom mapping
structure in order to specify which virtual ranks are mapped to a native rank. This is useful in cases where users desire to put all replicas on the same physical host to decrease network latency [23].

The two functions that RedMPI implements are **All-to-All** and **Message Plus Hash** (MsgPlusHash).

### 2.5.1 All-to-All

All-to-All, as the name indicates, sends all replicated messages to all replicated receivers. Thus, for a redundancy degree of three, each sender sends three messages where one message goes to each replica receiver as demonstrated by Figure 2.4.
4.2. The process of sending *degree* messages for each replica is achieved by interposing `MPI_ISEND` and determining all replicas for each virtual rank. When a MPI application receives a message, RedMPI waits for all replicas to finish during an `MPI_Wait` or an `MPI_Test` before verifying the data. The actual verification happens before `MPI_Wait` or `MPI_Test` return to the MPI application. Verification is performed via memory comparison or computing a hash of each replica and then comparing these hashes. With replication of two, in case of corruption, an error message is logged denoting error detection. With a replication degree of three or more, correct messages are determined by a simple voting majority. RedMPI then ensure that the MPI application’s receive buffer contains the correct data.
2.5.2 Message Plus Hash (MsgPlusHash)

MsgPlusHash introduces a major improvement over All-to-All as it decreases the number of total messages that need to be sent. In this case, only one copy of the message is sent in addition to a very small hash. This change allows each replica to send its data once with the hash being sent later. Thus, the total number of messages sent is $n \cdot r$; where $n$ is the number of messages sent and $r$ the replication degree, a major reduction from $n^r$. To detect a message corruption, the minimum requirement is comparison between two different sources since a receiver gets a copy of the message from a replica and a hash from another as shown in figure 2.6.

Each sender must calculate where to send its message and where to send a hash of it. Simply, the sender sends the actual message to the receiver with the same replica rank whereas the hash of the message is sent to the replica rank plus one.
If the destination rank for the hashed message exceeds the replication degree, the
destination wraps around to 0. Correction in this case is a multi-step process since a
receiver needs to determine whether the actual message or the hash is corrupted. In
the event of corruption, two adjacent receiving replicas will be aware of the corrupt
sender. However, the receiver cannot easily identify whether their message or hash was
corrupted. A handshake process happens between the adjacent replicas to determine
which of them holds a correct message. After the handshake, the replica with the
higher rank transmits a correction message to the lower ranked replica since the higher
rank will always contain the corrupt message with a bad hash.

2.5.3 Async

The Async method is the first asynchronous extension to RedMPI. This method
functions similarly to the previously mentioned AllToAll function. The main differ-
ences between the two functions is that Async provides asynchronous behavior using
P MPI_Waitany instead of MPI_Waitall in the MPI_Wait implementation of
RedMPI. Async uses copies of send buffers since a successful return from MPI_Wait
indicates that the send buffer could be reused. However, this is not the case with
Async since detection is performed before all replicas are received.

Async does not wait for all replicas to be received and therefore in order to handle
late arriving replicas, Async implements an integrity-verification function that makes
sure that all underlying requests have been completed. The integrity-verification func-
tionality is also used is MPI_Finalize after a call to P MPI_Waitall to capture all
incomplete requests. Figure 2.7 gives an overview of the Async method functionality.

Async does not add huge overhead compared to AllToall. The only additions
are the copy buffers for send requests, hash buffers to keep track of all previously
received replicas as detection happens asynchronously, and a couple of others variable to determine incomplete requests. Finally, the *integrity-verification* method adds a small amount of time to the execution but that is indeterminable since it depends on the amount of times this functions will have to be called before all requests are complete [24].
2.5.4 AsyncHash

The second asynchronous extension to RedMPI is AsyncHash which is similar to the MsgPlusHash function. The only difference between the two functions is that AsyncHash allows for messages to be sent and received without blocking the application.

Unlike Async, MsgPlusHash invokes PMPI_WAIT to wait for a full set of messages to be sent or received. This means that MsgPlusHash does not allow the application to complete faster than any set of processes since all replicas are dependent on each others based on the description of MsgPlusHash. This functions implements shadowing 6.1.3 to achieve faster execution time compared to MsgPlusHash.

Similarly to Async, the additional storage requirements are minimal such as buffers for locally computed hashes and an incomplete flag. The integrity-verification has the same considerations as Async’s.
Chapter 3

IMPLEMENTATION

Our work adds fault recovery to the asynchronous RedMPI functions. Based on the architecture of Async and AsyncHash discussed in 2.5.3 and 2.5.4 we present the viability of recovery in each. First, we discuss the challenges in the implementation of recovery for AsyncHash then we discuss Async's implementation which is different from the AllToAll recovery implementation.

The challenge in the implementation of recovery in the AsyncHash in an asynchronous fashion is caused by the nature of this function. Since the message is not sent to all replicas as shown in figure 2.6, the recovery of one replica is dependent on its adjacent replicas in order to figure out whether the message is corrupt or the hash is. The handshake process required to perform this correction needs all replicas to be received and therefore defeats the purpose of asynchronous detection.

On the other hand, the challenge in the implementation of recovery for Async is caused by the fact that it uses PMPI_Waitany instead of MPI_Waitall in the MPI_Wait implementation of RedMPI. Therefore processes with corrupted data that continue execution before a full set of messages is received will have to rollback to a state before corruption happened following a C/R scenario.

3.1 Async Recovery

In order to determine correct data when corruption happens, a voting algorithm is used. However, the problem the voting algorithm runs into is when for higher
degree replications the probability of multiple corrupted data agreeing with each other becomes possible. For example, for a replication degree of 4, there is a possibility, even if it is almost negligible, that a couple replicas will have similar values. In this case, no further conclusions can be made and the function would return a non successful value.

The voting proceeds by a creating a table that maps a previously calculated hash from detection to a number of occurrences. The hash with the biggest occurrence is assumed to contain the correct data. This data is then propagated to other replicas of the process. We considered first rolling back the application to a state before corruption happened to restart computation but saving the state of all replicas and restarting the application will defeat the whole purpose of asynchronous detection. Another approach that we also studied at is to only correct the data in the fast replica but this gives inconsistency since different replicas could receive or send data quicker than others at different points during execution. Therefore, in order to maintain relatively good performance with asynchronous detection, we suggest having the replica with rank 0 be the corrected process during the execution of a program with faults. This solution waits for process 0 to arrive and therefore also needs to have information about other replicas to determine whether a fault happened or not. Meanwhile if the other replicas had not arrived yet, replica 0 continues execution and once its value is verified, it either rolls back to a prior correct state or continues execution. At the end of each set of full messages, the value of replica 0 is propagated among all other replicas.

3.1.1 Edge Cases

The asynchronous behavior of Async was only implemented in MPI_Wait but not in MPI_Test nor in MPI_Testall. Therefore, we added regular recovery for those
functions. By *regular* we mean that recovery happened on the sport without the need of checkpointing and rolling back.

In addition the recovery depends on the type of application running. In 4.2 we show results for two different benchmarks that reflect the recovery mechanism used. The two types of applications are:

- Applications that do not terminate on corrupt values before correction and rollback. Such application will not try to access invalid memory or divide by zero in case of invalid data. We develop an addition testing benchmark 4.2.1 that does not terminate if corrupted data was received and computation happened on the receiving side before detection/recovery.

- Other applications must terminate when the detected failure produces a memory error, such as *segfault*. This types of applications cannot benefit from asynchronous detection since a fast corrupted replica that is received before being checked or corrected will break/kill the application. We look at matrix multiplication when evaluating our recovery algorithm and notice that whenever the receiver end gets a corrupted message that is the first replica will proceed with calculation and therefore cause the application to break. In order to fix this issue we had to wait not only for two replicas to arrive in order to detect faults but for the third as well to be able to recover from it since if either corrupted replicas continue the execution will cause the application to terminate. Unfortunately, that entails that this type of applications needs a more preventative detection and recovery functionality that could also be dependent on the user’s application. We will further discuss that in 7. By preventative we mean it should be either detected by user code or synchronous detection not asynchronous and therefore there will be a small overhead when this happens.
3.2 Fault Injector

The recovery process is verified using a fault injector that comes with RedMPI. However, multiple fixes were done to achieve correct behavior of this injector. The built-in fault injector in RedMPI is used to cause data corruption in order to check the validity of recovery. The fault injector is designed as such it takes in a given frequency $1/x$ during launch which is the probability of any single message becomes corrupted. The fault injector targets send buffers to ensure that it affects the execution of the program especially on the recipients side. Once the message to be corrupted is picked, a random bit is flipped. This process does not take into account the types of data that is being sent in the message. The fault injector will only count the total number of bits in the message then decide which random bit to flip. After taking a closer look on the code, we realize that the fault injector is broken and does not function properly.

First, the authors use a modulo operation between the value generated by calling the `random()` function and the $1/x$ frequency. The $1/x$ frequency is captured as a string then the `atoi()` function is used on that value to generate an `Int` value. This process basically makes the frequency equal to 1 every time. As for the condition for corruption, the authors have an `if` check for whether the modulo operation results a 0 which is always true considering that the frequency value is always 1 and any number module 1 equals 0. This is a problem since all replicas of messages are corrupted and therefore there is no possible way of recovery. We changed this to have the fault injector corrupt at most half of the replicas during the execution of an application with the corruption flag on.

The fault injector not only corrupted all replicas of a message but also corrupted data that reside in memory. The purpose of implementing recovery is correction of
data that is sent and received within MPI but is not responsible of fixing data that is outside of the scope of MPI operations. Therefore, corrupting data that is not used in \texttt{MPI\_Send} operations is out of the scope of our recovery process.
Chapter 4

EVALUATION

4.1 Detection Testing

We first evaluate the performance of the RedMPI synchronous and asynchronous detection in order to get a general idea of the advantages of asynchronous detection. This initial testing is done without fault injection since we are only interested in the detection scheme. The testing environment used for this evaluation is as follows.

- 30 nodes, each with a 14-core Intel(R) Xeon(R) CPU E5-2695 v3 @ 2.30GHz and 32GB of RAM. The nodes are connected via Ethernet using 1Gbs NICs. However, it is important to note that for one test case, (4.1.4), and recovery testing the Ethernet card was upgraded to 10Gbs.

In order to compare the performance between the asynchronous extensions and the synchronous functions, the NASA benchmarks were run under different environments. Four benchmarks were run: SP, BT, LU, and CG which will be detailed later [4]. The different classes specify the complexity and size of the benchmark. The benchmarks were run with varying number of processes that accommodates the benchmark’s requirements. It is important to note that the MPI environment uses a virtual number of processes so when saying that 32 or 64 processes are run on 30 nodes that means that MPI will distribute the load based on its default balancing algorithm.

Tables 4.1 and 4.2 illustrate the behavior of the RedMPI synchronous and asynchronous methods for benchmarks with varying process size. The data collected is the
average of 10 iterations for each value. Standard deviations are calculated to show the consistency of the data.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Procs</th>
<th>MsgPlusHash</th>
<th></th>
<th></th>
<th>AsyncHash</th>
<th></th>
<th></th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2x</td>
<td>SD</td>
<td>3x</td>
<td>SD</td>
<td>2x</td>
<td>SD</td>
<td>3x</td>
</tr>
<tr>
<td>SP(B)</td>
<td>32</td>
<td>142.3</td>
<td>8.85</td>
<td>174.99</td>
<td>2.4</td>
<td>113.4</td>
<td>7.95</td>
<td>132.14</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>177.52</td>
<td>7.22</td>
<td>215.02</td>
<td>1.97</td>
<td>152.39</td>
<td>10.63</td>
<td>177.73</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>225.96</td>
<td>6.89</td>
<td>288.35</td>
<td>3.69</td>
<td>215.38</td>
<td>27.37</td>
<td>241.61</td>
</tr>
<tr>
<td>BT(B)</td>
<td>32</td>
<td>92.35</td>
<td>1.14</td>
<td>106.23</td>
<td>1.2</td>
<td>79.71</td>
<td>1.61</td>
<td>95.21</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>96.79</td>
<td>1.02</td>
<td>111.52</td>
<td>1.45</td>
<td>85.93</td>
<td>2.5</td>
<td>98.63</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>120.92</td>
<td>1.18</td>
<td>148.73</td>
<td>1.1</td>
<td>112.68</td>
<td>1.02</td>
<td>130.21</td>
</tr>
<tr>
<td>LU(B)</td>
<td>32</td>
<td>26.56</td>
<td>2.98</td>
<td>50.9</td>
<td>3.05</td>
<td>32.51</td>
<td>3.29</td>
<td>47.39</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>52.96</td>
<td>1.85</td>
<td>79.55</td>
<td>3.72</td>
<td>51.62</td>
<td>0.9</td>
<td>58.88</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>90.27</td>
<td>3.55</td>
<td>130.25</td>
<td>7.48</td>
<td>71.67</td>
<td>0.99</td>
<td>76.67</td>
</tr>
<tr>
<td>CG(C)</td>
<td>32</td>
<td>20</td>
<td>4.39</td>
<td>34.51</td>
<td>0.81</td>
<td>18.94</td>
<td>0.41</td>
<td>19.93</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>18.46</td>
<td>4.17</td>
<td>44.81</td>
<td>8.67</td>
<td>16.64</td>
<td>0.32</td>
<td>17.74</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>17.18</td>
<td>9.39</td>
<td>85.37</td>
<td>27.47</td>
<td>13.14</td>
<td>0.49</td>
<td>16.01</td>
</tr>
</tbody>
</table>

Table 4.1: Execution time in seconds for NPB using the MsgPlusHash and AsyncHash functions
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Procs</th>
<th>AllToAll</th>
<th></th>
<th></th>
<th>Async</th>
<th></th>
<th></th>
<th>Improvement</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>2x</td>
<td>SD</td>
<td>3x</td>
<td>SD</td>
<td>2x</td>
<td>SD</td>
<td>3x</td>
<td>SD</td>
</tr>
<tr>
<td>SP(B)</td>
<td>32</td>
<td>250.06</td>
<td>15.92</td>
<td>546.61</td>
<td>39.68</td>
<td>115.9</td>
<td>0.56</td>
<td>130.66</td>
<td>0.52</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>314.79</td>
<td>12.71</td>
<td>646.27</td>
<td>25.88</td>
<td>133.01</td>
<td>0.91</td>
<td>153.29</td>
<td>0.84</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>268.94</td>
<td>8.81</td>
<td>586.23</td>
<td>35.76</td>
<td>135.32</td>
<td>5.91</td>
<td>176.76</td>
<td>3.65</td>
</tr>
<tr>
<td>BT(B)</td>
<td>32</td>
<td>178.38</td>
<td>15.8</td>
<td>412.38</td>
<td>30.97</td>
<td>66.59</td>
<td>3.05</td>
<td>81.98</td>
<td>0.50</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>157.62</td>
<td>5.84</td>
<td>338.97</td>
<td>16.79</td>
<td>64</td>
<td>1.19</td>
<td>77.62</td>
<td>0.39</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>195.38</td>
<td>8.28</td>
<td>363.41</td>
<td>11.31</td>
<td>77.45</td>
<td>1.09</td>
<td>106.24</td>
<td>0.32</td>
</tr>
<tr>
<td>LU(B)</td>
<td>32</td>
<td>83.15</td>
<td>2.35</td>
<td>162.12</td>
<td>8.64</td>
<td>28.04</td>
<td>0.45</td>
<td>37.99</td>
<td>2.00</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>111.22</td>
<td>4.93</td>
<td>227.75</td>
<td>8.45</td>
<td>34.6</td>
<td>0.42</td>
<td>39.66</td>
<td>1.85</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>177.2</td>
<td>5.92</td>
<td>335.09</td>
<td>17.55</td>
<td>53.47</td>
<td>1.14</td>
<td>57.71</td>
<td>3.00</td>
</tr>
<tr>
<td>CG(C)</td>
<td>32</td>
<td>36.14</td>
<td>0.29</td>
<td>41.05</td>
<td>8.52</td>
<td>20.07</td>
<td>0.49</td>
<td>27.24</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>36.94</td>
<td>1.58</td>
<td>64.07</td>
<td>14.97</td>
<td>19.33</td>
<td>0.24</td>
<td>30.19</td>
<td>1.03</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>46.21</td>
<td>9.02</td>
<td>97.7</td>
<td>50.56</td>
<td>15.38</td>
<td>0.09</td>
<td>30.02</td>
<td>2.27</td>
</tr>
</tbody>
</table>

Table 4.2: Execution time in seconds for NPB using the AllToAll and Async functions
In order to take a closer look at the results, we use the following graphs to show the runtime data comparisons between the asynchronous and synchronous RedMPI functions for various benchmarks. We also discuss the properties of each benchmark as well as its performance. The general trend that can be seen is that the asynchronous methods for both AlltoAll and MsgPlusHash perform better than their synchronous counterparts. They also show a smaller variation for the number of replicas used during calculation.

4.1.1 SP(B)

Scalar Penta-diagonal solver (SP) is a CFD application. Multiple set of multiple independent systems of nondiagonally dominant, scalar pentadiagonal equations are solved. A complete solution of the SP benchmark requires 400 iterations [4]. The results shown in figures 4.1 and 4.2 show that asynchronous detection is faster for both functions using double and triple redundancy.

![Figure 4.1: Runtime measurements for SP(B) benchmark using MsgPlusHash and AsyncHash](image_url)
4.1.2 BT(B)

The Block Tridiogonal (BT) benchmark is also a CFD application. In this benchmark, multiple independent systems of non-diagonally dominant, block tridiagonal equations with a 5x5 block size are solved. SP and BT are very similar but there is a fundamental difference with respect to the communication to computation ratio [4]. This shows clearly in the data collected as runtime of BT is noticeably faster than SP since it performs less communication compared to SP which decreases the time spent doing checks for faults. Same behavior is seen in figures 4.3 and 4.4 as asynchronous detection provides clear improvement in total runtime.
Figure 4.3: Runtime measurements for BT(B) benchmark using MsgPlusHash and AsyncHash

Figure 4.4: Runtime measurements for BT(B) benchmark using AllToAll and Async
4.1.3 LU(B)

The lower-upper diagonal (LU) benchmark is the third CFD application in the NPB sets. It does not perform a LU factorization but instead employs a symmetric successive over-relaxation (SSOR) numerical scheme to solve a regular-sparse, block 5x5 lower and upper triangular system. A complete solution of the LU benchmark requires 250 iterations. The improvement in runtime is also remarkable for this benchmark as show in figure 4.5 and 4.6.

Figure 4.5: Runtime measurements for LU(B) benchmark using MsgPlusHash and AsyncHash
Figure 4.6: Runtime measurements for LU(B) benchmark using AllToAll and Async

4.1.4 CG(C)

The Conjugate Gradient (CG) benchmark is used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication and employs sparse matrix-vector multiplication. A complete solution requires 75 iterations [4]. This benchmark was run on the 30 nodes after the NIC upgrade. This allowed us to run a more complex class, C. Initially, CG(C) would take close to 30 mins to complete with a 1GHz NIC. We can conclude the communication heavy aspect of this benchmark in this case. More importantly, figures 4.7 and 4.8 show the improvement in performance between asynchronous and synchronous detection.
Figure 4.7: Runtime measurements for CG(C) benchmark using MsgPlusHash and AsyncHash

Overall, asynchronous detection performs better than synchronous functions. In many cases Async detection, the asynchronous version of AllToAll has smaller runtimes even...
with a three degree replication compared to a two degree replication using *AllToAll* as show in figures 4.4, 4.8, 4.6, and 4.2.

**4.2 Recovery Testing**

As we discussed in 4.3, two different recovery functions were implemented to support the asynchronous behavior based on the application being run. We present two benchmarks to prove the correctness of the implemented recovery. Unit testing was also performed on the recovery function itself, at the end of recovery correct values were present in all replicas.

**4.2.1 Addition Benchmark**

Recovery testing is done using a simple yet what could be a compute heavy testing benchmark. The testing benchmark (we call it *addition benchmark*) takes in a value which is an exponent \( \text{exp} \) used to get a total number of chunks equal to \( 10^{\text{exp}} \). Once the value is calculated, every process other than the master process will add up \( 1/\text{total} \) a total times to get 1.

In figure 4.9 we can see from the initial debug code that the detection mode used is Async and the corruption frequency is set to 1 due to the nature of the fault injector which is detailed in 3.2. The replication degree is three which allows recovery to happen and the number of processes used is 2 indicated by the *MRMPI_VSIZE* variable. The reason we choose one slave process to do the computation in here is to be able to capture the output for the different phases; before and after corruption and after correction. The test framework can be easily scaled by changing number of process and/or the \( \text{exp} \) parameter. Figure 4.9 shows the values being added before
corruption, after corruption, and after recovery. We also see that the one replica 4.10 finished with the correct answer before the two others 4.11 which is due to the nature of the recovery detailed in 4.3.

We run the addition benchmark with an exponent value of 5 in the environment described in 4.1. Tables 4.3 and graph 4.12 illustrate the behavior of the the Async function with and without recovery at varying process size. The data collected is the average of 10 iterations for each value. Standard deviations are calculated to show the consistency of the data.

Figure 4.9: Async correction
Figure 4.10: First replica exiting

Figure 4.11: Second and third Replica exiting

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Procs</th>
<th>Async</th>
<th>Async with Recovery</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3x SD</td>
<td># corrupt messages</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>SD</td>
<td></td>
</tr>
<tr>
<td>Addition</td>
<td>32</td>
<td>36</td>
<td>6.2</td>
<td>38.5</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>41.6</td>
<td>2.8</td>
<td>46.6</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>68.3</td>
<td>10.5</td>
<td>82.7</td>
</tr>
</tbody>
</table>

Table 4.3: Execution time in seconds for the Addition benchmark using Async and Async recovery

Figure 4.12: Runtime measurements for the addition benchmark using Async and Async Recovery
From figure 4.12 we can observe that the overhead added by recovery is minimal. We also notice that this overhead is proportional to the amount of processes running which is intuitive since more processes means more failure and more time needed for recovery. We present the average slowdown for each case. We also count the average number of corrupt messages that were recovered during the execution of the benchmark.

4.2.2 Matrix Multiplication

We use the same environment described in 4.1 and run a simple matrix multiplication program that takes in input from the user to determine the width and height of the matrix. The matrices are generated using random float between 0 and 1. Tables 4.3 and graph 4.12 illustrate the behavior of the the Async function with and without recovery at varying process size. The data collected is the average of 10 iterations for each value. Standard deviations are calculated to show the consistency of the data.

<table>
<thead>
<tr>
<th>Size</th>
<th>Procs</th>
<th>Async 3x</th>
<th>SD</th>
<th>Async with Recovery 3x</th>
<th>SD</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000x5000</td>
<td>32</td>
<td>51.09</td>
<td>2.33</td>
<td>63.78</td>
<td>2.33</td>
<td>24.84</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>54.05</td>
<td>2.02</td>
<td>71.12</td>
<td>1.14</td>
<td>31.58</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>60.4</td>
<td>5.14</td>
<td>83.8</td>
<td>5.03</td>
<td>38.74</td>
</tr>
</tbody>
</table>

Table 4.4: Execution time in seconds for the matrix multiplication benchmark using Async and Async recovery
Figure 4.13: Runtime measurements for the matrix multiplication benchmark using Async and Async Recovery

Table 4.4 and figure 4.13 show the disadvantage of waiting for all replicas in order to achieve recovery in this situation. The slowdown is more noticeable than the results show in 4.3. Therefore, the benefits of an asynchronous detection decrease when dealing with such applications. We also notice the increase in overhead when the number of processes increase which is expected as mentioned in the previous case. We discuss how to gain the most of asynchronous detection in this case in 7.

Finally, after observing recovery in both scenarios we show that this could be achieved in various application with a small overhead. As expected, the first type of recovery ran with a smaller overhead than the second type. Overall, the results show that recovery could be added to asynchronous detection without immensely affecting the overall performance.
While working on this project we determine a couple of important things to discuss about the current implementation of this library. The most important subject is the implementation of \texttt{MPI\_Test}. \texttt{MPI\_test} as introduced in the MPI library is as follows \cite{2}.

\begin{verbatim}
int MPI_Test (MPI_Request *request, int *flag, MPI_Status *status)
\end{verbatim}

\texttt{request} is the handle of the communication request, \texttt{flag} identifies whether the operation is complete or not, and \texttt{status} carries information about the active operation. A call to \texttt{MPI\_Test} returns \texttt{flag = true} if the operation identified by request is complete. The call returns \texttt{flag = false}, otherwise.

The implementation of the \texttt{MPI\_Test} in the RedMPI library does not function as expected. Basic testing using this function was implemented to prove so. We look into this since \textit{Async} only implements \texttt{PMPI\_Waitany} in \texttt{MPI\_Wait} and not in \texttt{MPI\_Test} as mentioned in 2.5.3. First there was no documentation stating that the implementation of \texttt{MPI\_Test} in RedMPI is functional. In addition none of the examples provided with the library calls the \texttt{MPI\_Test} function. It is also important to note that \texttt{MPI\_Test} could potentially have more advantages over \texttt{MPI\_Wait} with asynchronous detection since a call to \texttt{MPI\_Test} does not wait until the request is complete before proceeding to others unlike \texttt{MPI\_Wait}. 
Chapter 6

RELATED WORK

6.1 Redundancy

Redundancy has been significantly researched over the years and many applications implemented this solutions in different areas. We present the most important works that implemented redundancy as a resilience solution.

6.1.1 Masking Redundancy

Masking redundancy uses extra components to mask the effect of faults. The idea of masking redundancy was first introduced as solution to fault detection in hardware. Thus, faulty components are masked by the presence of other components. The most general masking technique is the triple modular redundancy (TMR). Three identical copies as input provide a voting element with each a separate result. The voter accepts outputs from all three modules and produces a majority vote as its output. Modular redundancy has been used in different field such as aerospace, information technology, and command and control [52]. Other modular redundant uses include Dynamic and Scalable Dual Modular Redundancy (DDMR) which addresses the problem of soft errors in chip multiprocessors [28].

6.1.2 Redundant Execution of MPI Applications

Multiple libraries that used redundant execution for MPI applications have been developed.
rMPI uses the MPI profiling interface (PMPI) for interpositioning. rMPI replicates nodes while execution and in case of a node failure, the replicated node continues without interruption. The application fails when two replicas fail. rMPI enables redundant computation and uses two separate consistency protocols to maintain consistency between rank replicas [20, 21].

The modular-redundant Message Passing Interface (MR-MPI) is a similar solution for transparent HPC redundancy via PMPI interpositioning. In MR-MPI, a redundantly executed application runs with \( r \cdot m \) native MPI processes, where \( r \) is the number of MPI ranks visible to the application and \( m \) is the replication degree. Messages are replicated between redundant nodes. For low-level, point-to-point benchmarks, the impact can be as high as the replication degree [30, 18]. The implementation of RedMPI extends on this work by adding the \( MsgPlusHash \) function for better internal communication as well as adding SDC protection [23].

PaRep-MPI is another MPI prototype developed as a fault-tolerant solution. PaRep-MPI implements a mechanism for proactive fault tolerance using partial replication of a set of application processes. This fault tolerance framework adaptively changes the set of replicated processes periodically based on failure predictions to avoid failures [27].

VolpexMPI is an MPI library implemented from scratch. It is designed to enable seamless forward application progress in the presence of frequent node failures as well as dynamically changing networks speeds and node execution speeds. This solution does not provide SDC correction, however, it provides good performance as replication protocols are part of the low-level communication inside the MPI library. VolpexMPI is implemented to be only used in a distributed environment [39].

Fault Tolerant Messaging Interface (FMI), which enables extremely low-latency
recovery, uses a survivable communication runtime coupled with fast, in-memory C/R, and dynamic node allocation. FMI provides message-passing semantics similar to MPI, but applications written using FMI can run through failures. The FMI runtime software handles fault tolerance, including check pointing application state, restarting failed processes, and allocating additional nodes when needed [48].

6.1.3 Shadowing

Shadow computing is a new computational model, which provides goal-based adaptive resilience through the use of dynamic execution. Using this general model, shadow replication enables a parameterized tradeoff between time and hardware redundancy to provide fault tolerance. Adaptive Resilience is the ability of the system to dynamically harness all available resources to achieve the highest level of QoS (Quality of Service) for a given application [43].

Lazy Shadowing, is an efficient scalable approach to achieve high-levels of resilience, through forward progress in failure-prone computing environments. Lazy Shadowing associates with each process a shadow (process) that executes at a reduced rate. During failure recovery, each shadow rolls forward to catch up with its leading process. the goal of shadowing is mainly to minimize energy consumption [15, 14].

6.1.4 Cloning

Live process cloning is another MPI resilience solution that follows the same idea as replication. The main difference between cloning and replication is that process cloning happens on-the-fly when needed for failure recovery. Therefore, such systems are based on forward execution rather than rollback to checkpoints. Work done by Mueller et al. proves that the reliability of a dual redundant system with live process
6.1.5 Redundant Multithreading

Another area of work similar to replication is Redundant Multithreading (RMT) used as a fault-detection approach for microprocessor cores. RMT runs two identical processes as independent threads and compares their outputs for mismatch. This work was based on the previously published Simultaneous and Redundantly Threaded (SRT) processor that provides transient fault coverage [44, 45].

Adaptive RMT is another approach that specifies particular code blocks to be
executed redundantly. This flexible execution provides huge improvement in performance compared to full redundancy. Users can determine the correctness of the application based on the errors detected in parts of the running program. Later, lazy fault detection was added to further optimize the use of redundancy by prioritizing the application’s primary computation over the fault detection. Hukerikar et. al. implement a system that relaxes the requirement that the redundant threads synchronize and compare results immediately. Results show lower time to solution over adaptive RMT for a range of scientific computational kernels [33, 62]. Figure 6.1 shows the difference between the initial adaptive RMT and aRMT with added lazy fault detection.

Additional improvements to the basic RMT approach were implemented using a checkpoint-based RMT (C-RMT) followed by asynchronous C-RMT. C-RMT simply implement checkpoints where master processes are compared to slave processes for fault detection. Of course, this asynchronous behavior imposes the downside of waiting for all processes to reach the same position. AC-RMT fixes this issue by having two context storage areas for each thread, one for detecting faults, and the other for saving the last checkpoint used for fault restoration [55, 51].

Besides hardware level RMT, a compiler-level RMT was introduced by So et al. EXPERT is a hardware fault detection scheme in all hardware components. EXPERT runs a checker thread for the main execution thread. The separate threads run on two different cores of a multi-core processor. After each memory write committed by the main thread, the checker thread compares that data to its locally computed values for error detection [45].
6.2 Additional Resilience Solutions

Besides C/R and redundancy, other solutions were developed for resilience. Some of those techniques are presented below.

6.2.1 ULFM

User-level failure mitigation (ULFM) interface has been proposed to provide fault-tolerant semantics in MPI. Contrary to RedMPI and C/R, ULFM exposes fault-tolerance capabilities to the application such that users have to incorporate this functionality into their applications [6]. ULFM is suitable for masterworker applications but it provides few benefits for more common bulk synchronous MPI applications [37]. Other evaluations of ULFM in MPI also show that it has little to no impact on performance and produces satisfactory recovery times when there are failures [7]. ULFM was also implemented to work with MPICH, a popular open source MPI implementation [8].

6.2.2 Fenix

Fenix is a framework built using the idea of ULFM within MPI for process and data recovery, Fenix does not detect faults but assumes that errors are indicated via error codes during runtime which is provided by the use of ULFM with MPI. When faults occur, Fenix repairs communicators transparently then returns execution control to the application. Process recovery relies on the Fenix_Init function which initializes the library, specifies extra resources in case of rank failure, creates a logical resumption point in case of rank failure. Data recovery relies on Fenix’s redundant storage but users can also chose to use external libraries such as SCR (Scalable Checkpoint
6.2.3 Local Failure Local Recovery

Local Failure Local Recovery (LFLR) provides the ability to recover locally and continue application execution when a process is lost. LFLR permits local recovery for a local failure to keep the remaining processes alive during the recovery. LFLR uses MPI-ULFM for detection which sends back error messages when failure occurs. Spare processes are provided to take over failed ones. Therefore, the recovery process follows three steps: process recovery, data recovery, and application state recovery [58].

6.2.4 Algorithm-Based Fault Tolerance

Algorithm-Based Fault Tolerance (ABFT) was initially suggested as a means for designing fault tolerant parallel systems especially for VLSI processor arrays. ABFT’s performance was addressed and compared to C/R in the work done by Kabir et al. [34]. ABFT was also tested to work with MPI. When failure occurs during the execution of applications, the failed node is replaced with the corresponding redundant node to continue the execution. At the end of execution, the correct solution can be recovered algorithmically [60]. The ABFT method has been studied in many applications, such as matrix multiplication [57], ScaLAPACK [13], HPL [16], and iterative methods in PETSc [12].
We have added lazy fault recovery to the Async function and explained in which applications this functionality is not viable. During this work, we encountered many issues that had to be fixed such as the fault injector. We present two benchmarks that show the success of recovery. We conclude that the overhead is not very significant in application where asynchronous recovery is possible but we notice an increasing overhead when asynchronous detection is not applicable. We also present a validation of previous work to show that asynchronous detection is in fact better that synchronous detection in order to conclude about the overall performance of asynchronous detection and recovery.

The major area of future work is to implement an asynchronous recovery that would work with any type of application which might be impossible based on the nature of this library and asynchronous detection implementation. As discussed earlier, this becomes highly dependent on user’s testing benchmarks. Users can therefore have sanity checks on data and in case they would cause a termination of the program, they would either stall until a corrected value is received or raise a MPI flag during execution to speed up detection thus recovery. Another solution we could look at is specifying at runtime how critical faults are to the application which will determine what recovery will be used.


[8] W. Bland, K. Raffenetti, and P. Balaji. Simplifying the recovery model of


[22] D. Fiala, F. Muller, C. Engelman, K. Ferreira, R. Brightwell, and R. Riesen. Detection and correction of silent data corruption for large-scale


[61] K. L. Wu and W. K. Fuchs. Recovered distributed shared virtual memory: 
Memory coherence and storage structures. In Proceedings of the Nineteenth 

multithreading architecture. In 2010 IEEE 16th Pacific Rim International 