LAZY FAULT DETECTION 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

Corey Ford

June 2016

COMMITTEE MEMBERSHIP

TITLE:	Lazy Fault Detection for Redundant MPI
AUTHOR:	Corey Ford
DATE SUBMITTED:	June 2016
COMMITTEE CHAIR:	Chris Lupo, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Clint Staley, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Phillip Nico, Ph.D. Professor of Computer Science

ABSTRACT

Lazy Fault Detection for Redundant MPI

Corey Ford

As the scale of supercomputers grows, it is becoming increasingly important for software to efficiently withstand hardware and software faults. Process replication is one resilience technique, but typical implementations require replicas to stay closely synchronized with each other. We propose algorithms to lazily detect faults in replicated MPI applications, allowing for more flexibility in replica scheduling and potential power savings. Evaluation shows that, when all processes are operated at full power, this approach allows applications to complete substantially faster as compared to using a synchronized model, and often as fast as in non-replicated execution.

ACKNOWLEDGMENTS

Many thanks to:

- Chris Lupo, for excellent guidance on this thesis
- David Fiala and Frank Mueller, for help getting started with RedMPI and valuable feedback on this work
- My parents, for their love and support

I would like to thank Sandia National Laboratories for supporting this work. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under contract DE-AC04-94AL85000. This work was funded by NNSA's Advanced Simulation and Computing (ASC) Program.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

High-performance computing uses large clusters to perform complex computations. Current top supercomputers can operate at petascale, on the order of 10^{15} FLOPS, and exascale systems with the capacity for 10^{18} FLOPS are anticipated within the next decade.

As scale increases, however, so does the number of system components that can fail and disrupt application execution. Since the frequency of faults for a single component remains relatively constant, the probability of having no failures in the entire cluster during a given time period approaches zero as the cluster grows. At current scale, the mean time between failures (MTBF) of a system may be on the order of hours or less. Typical scientific HPC applications may have execution times measured in days and minimal tolerance for error. Understanding and efficient mitigation of failure are therefore becoming crucial. Proper and efficient application execution in spite of various faults, including *silent data corruption* (SDC) to memory, is the focus of the subfield of *resilience*, which encompasses a wide range of possible solutions.

The most common and longstanding approach to failure mitigation is checkpoint/restart, which periodically saves application state to reliable storage so that it can be recovered in the event of a failure in the system. However, the resource demands of checkpointing scale linearly with the size of the cluster, and increasingly complex optimizations are required to keep checkpointing time less than MTBF.

As an alternate approach, replication performs redundant computations on multiple nodes. The state of a process and its replicas are kept synchronized so that a replica can take over in case of a failure on the primary node. Additionally, replication

can provide detection and potentially correction of transient errors on any node, such as silent data corruption. In a naive approach, replication requires double or triple the hardware of an ordinary execution. More efficient solutions can predict when and where replication is necessary, or perform approximate or slower-paced computations on a replica. Unfortunately, synchronization between replicas to detect SDC introduces inefficiencies and requires them to execute at the same speed.

In this work we demonstrate the utility of lazy fault detection in an HPC context. Lazy fault detection, where corrupted data is allowed to be identified as such after it has been used, improves the efficiency and flexibility of process replication while retaining the ability to detect SDC. We develop a practical implementation of lazy fault detection for replicated MPI programs, and show that it generally performs better than an equivalent synchronized approach across varied benchmark workloads.

The remainder of this thesis is organized as follows. Chapter 2 introduces the MPI system along with the problem of resilience and its various solutions. Chapter 3 develops the design of lazy fault detection within a software library, and Chapter 4 discusses our implementation. The performance of our implementation is evaluated in Chapter 5. Chapter 6 reviews related work in resilience. Finally, Chapter 7 concludes and discusses directions for future work.

Chapter 2

BACKGROUND

2.1 MPI

The Message Passing Interface (MPI) is a standardized interface providing communication and other capabilities to parallel programs. The MPI standard defines APIs in C and Fortran; implementations typically also provide C++ interfaces, and bindings to many other languages are available.

2.1.1 Operations

The features of MPI discussed in this thesis are available in version 1 of the MPI standard [1], and do not include additional features introduced in later versions [2, 3]. Some common MPI operations are listed (with C function prototypes as defined in Open MPI 1.6) in Figure 2.1. All operations return a status code to signal error. Communication operations are performed with respect to a *communicator* based on a subset of processes; a predefined global communicator suffices for many applications, so we do not discuss the `MPI_Comm` parameters further.

When an MPI application is launched, each process is assigned a unique integer *rank* starting from 0. Operations pertaining to the runtime configuration are used in practically every MPI application:

- `MPI_Init` initializes the MPI execution environment (and may ensure that all processes have the same `argv` vector). It must be called before any other MPI operations.
- `MPI_Finalize` cleans up the MPI execution environment. It must be called

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm);
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Comm_size(MPI_Comm comm, int *size);
int MPI_Finalize(void);
int MPI_Init(int *argc, char ***argv);
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
              int source, int tag, MPI_Comm comm,
              MPI_Request *request);
int MPI_Isend(void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm,
              MPI_Request *request);
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status);
int MPI_Reduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype, MPI_Op op, int root,
               MPI_Comm comm);
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm);
int MPI_Test(MPI_Request *request, int *flag,
             MPI_Status *status);
int MPI_Waitall(int count, MPI_Request *array_of_requests,
                MPI_Status *array_of_statuses);
int MPI_Waitany(int count, MPI_Request *array_of_requests,
                int *index, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

Figure 2.1: C prototypes of some common MPI functions

after all other MPI operations have completed.

- `MPI_Comm_size` determines the total number of processes.
- `MPI_Comm_rank` determines the calling process's individual rank.

There are several core MPI operations providing point-to-point communication:

- `MPI_Isend` starts a nonblocking send to another process. The rank of the receiving process is specified, along with the memory location, size, and type of a buffer to send. This initializes an `MPI_Request` object.
- `MPI_Irecv` starts a nonblocking receive from another process. The rank of the sending process is specified, along with the memory location, size, and type of a buffer to be filled with the received message. This initializes an `MPI_Request` object.
- `MPI_Wait` takes a previously initialized `MPI_Request` object and blocks until the operation is complete. For send requests, a return from `MPI_Wait` guarantees that the buffer is no longer needed and can be reused or deallocated by the application. For receive requests, a return from `MPI_Wait` indicates that the buffer has been filled with a received message.
- `MPI_Test` takes a previously initialized `MPI_Request` object and sets a flag to indicate whether the operation is complete. A return from `MPI_Test` with `*flag == 1` gives the same guarantees as a return from `MPI_Wait`.

The semantics of many other MPI operations that initiate or determine the status of communications can be defined in terms of those already mentioned. For instance:

- `MPI_Send` is a blocking send, equivalent to `MPI_Isend` followed by `MPI_Wait` on the same `MPI_Request` object.

- `MPI_Recv` is a blocking receive, equivalent to `MPI_Irecv` followed by `MPI_Wait` on the same `MPI_Request` object.
- `MPI_Waitall` waits for all requests in an array to complete, equivalent to `MPI_Wait` on each in turn.
- `MPI_Waitany` waits for at least one request in an array to complete, and could be implemented in terms of `MPI_Test`.
- `MPI_Testall` and `MPI_Testany` perform the analogous status-testing operations.
- `MPI_Bcast` and `MPI_Reduce` are examples of collective operations that send to or receive from all processes. These can be defined in terms of individual point-to-point communications with each process.

2.1.2 Execution

There are multiple implementations of MPI, including Open MPI [21], MPICH, and proprietary implementations from Intel and others. An MPI implementation consists of several components. The first of these is a library (and accompanying header files) implementing the standard interface against which MPI applications can build. An MPI implementation also often provides compiler wrappers (such as an `mpicc` command) to easily compile and link an application with this library. The other major component is a set of runtime tools for launching an application binary as multiple processes, potentially on multiple host machines.

A small C program using nonblocking MPI communication is shown in Figure 2.2. This is intended to be executed using two processes, and will send a value from the first to the second before printing it at each process. A typical process (with Open MPI) to build such an application and execute it using two processes on two hosts is shown in Figure 2.3. Using this process, two instances of the binary will be launched,

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

int main(int argc, char **argv) {
  int rank;
  int size;
  int value = 0;
  MPI_Request req;

  if (argc != 2) {
    printf("Usage: mpi_example num\n");
    return 1;
  }

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if (size != 2) {
    printf("Number of processes must be 2\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
  }

  if (rank == 0) {
    sscanf(argv[1], "%d", &value);
    MPI_Isend(&value, 1, MPI_INT, 1,
              0, MPI_COMM_WORLD, &req);
  } else {
    MPI_Irecv(&value, 1, MPI_INT, 0,
              0, MPI_COMM_WORLD, &req);
  }

  MPI_Wait(&req, MPI_STATUS_IGNORE);
  printf("Value at %d is %d\n", rank, value);
  MPI_Finalize();

  return 0;
}
```

Figure 2.2: Example MPI program using nonblocking communication

```
$ mpicc -o mpi_example mpi_example.c
$ mpirun -np 2 -host host1,host2 ./mpi_example 42
```

Figure 2.3: Process to build and execute example program

typically one on each host (subject to local configuration) and each with a unique process rank in $\{0, 1\}$.

2.2 Resilience

Resilience in HPC is a complex research area [8, 15]. A wide variety of different approaches have been developed.

We first introduce some necessary terminology for precisely defining the problem of resilience, following a well-established taxonomy [4].

- A *fault* is an erroneous hardware behavior. Faults include transistor malfunctions and memory bits that flip state or remain stuck. *Silent data corruption* (SDC) is a specific class of fault.
- An *error* is an erroneous system state caused by a fault. Errors include incorrect values stored in registers or memory.
- A *failure* is an erroneous system behavior resulting from an error. Failures include incorrect results (*Byzantine* failures) and node crashes (*fail-stop* failures).

Given these definitions, resilience is approximately synonymous with *fault-tolerance*, with the goal of preventing failures or mitigating their impact. While faults cannot be prevented in general, techniques such as ECC memory can prevent some faults from silently causing errors. Software checksums might catch errors before they cause failures. When these resilience measures are unsuccessful, failures will remain and must be addressed for the benefit of the application.

Substantial work has been done to analyze the faults that occur in real HPC systems and their components. Analysis of logs in production HPC environments give statistics on the frequency of various types of errors [10, 44, 24, 47, 45, 38].

Characterization of individual element failures can also be used in resource allocation to minimize faults [9].

2.3 Checkpointing

Checkpointing (commonly, *checkpoint/restart*) is a simple and long-standing resilience technique. In checkpointing, application state is periodically saved to persistent storage; when a fault is detected, the state is restored from the most recent checkpoint and execution continues.

The concept of checkpointing can be traced back several decades [32]. More recently, Elnozahy provides a good survey [16]. One widely used implementation of checkpointing is Berkeley Lab Checkpoint/Restart (BLCR) [25], which uses a Linux kernel module to save process state to a filesystem and integrates with MPI implementations to coordinate checkpointing across nodes.

The efficiency of checkpointing is a significant concern. If coordinated checkpointing is performed at the level of the entire system, both the frequency of failures and the resources required to store a checkpoint increase linearly with the number of nodes. With sufficiently many nodes, then, most time will be spend on checkpointing, restarting, and recomputing lost work. Efficiency can be improved somewhat through incremental checkpointing approaches and compression of checkpoint storage [18]. The optimal checkpointing interval has been derived in terms of the cost and frequency of checkpointing [50, 14].

2.4 Replication

Replication (or redundancy) is another common fault-tolerance approach. Classically, *state machine replication* considers the program as a deterministic state machine and

executes replicated copies of it [43, 33].

More generally, communicating processes can be replicated by introducing special handling of their communications. This results in r real processes being mapped to one "virtual" process, with a shared identity but with distinct replica ranks that are hidden from the application. Processes sharing the same replica rank form a replica set that mirrors the equivalent non-replicated system. In doing so, replication can handle failures more seamlessly than checkpointing, at the cost of generally requiring double or triple the computing resources of a single job. Replication appears more viable than checkpoint/restart for exascale systems [19].

Considering only fail-stop failures, it may suffice to simply run replica processes independently. For Byzantine failures, regular synchronization is necessary. To detect SDCs, assuming that processes execute deterministically, either the messages sent by replica processes can be compared, or their local state can be compared periodically.

RedMPI [20] is a library that implements replication with SDC detection via message comparison for MPI applications. When an application linked with RedMPI calls `MPI_Isend` or `MPI_Irecv`, messages are sent to or received from several replicas of the target virtual process. When the application calls `MPI_Wait` on a receive request, the library waits for all messages to be received and compares them for consistency, choosing the majority value to return to the application if possible.

Unfortunately, this comparison introduces synchronization between replica processes, requiring them to execute at the same speed and in lockstep. Lazy (asynchronous) comparison of messages would provide similar fault-detection benefits while reducing synchronization and allowing for more flexibility in replica process execution.

Chapter 3

DESIGN

Our primary goal in lazy fault detection for MPI applications is to avoid the need for a message receiver to wait for multiple sending nodes, as in RedMPI. This enables the receiver to proceed as soon as one message is received, avoiding further delay in the common case that this message is not corrupted. We assume the same model of message corruption proposed for RedMPI: SDC affects MPI send buffers, or other values in memory from where corruption eventually propagates to send buffers.

This model allows for combining SDC detection with shadow computing [36], where only the primary replica processes proceed at full speed and others can be operated at lower power. Slower replicas would provide eventual confirmation of results; additionally, if two replica processes complete in agreement, any further replicas can be terminated entirely. Lazy fault detection may even (subject to network limitations) allow an application to complete faster than without replication, since only the fastest replica response is required at each step.

As requirements, this design must preserve the expected semantics of MPI calls and error correction:

1. If `MPI_Recv` or `MPI_Wait` returns, or `MPI_Test` indicates completion, the application's buffer must contain a valid received message.
2. If, subsequently, the value that was placed in the application's buffer is determined to be incorrect, corrective action must be taken.
3. The application cannot exit until all messages have been verified.

With asynchronous fault detection, any application call to `MPI_Recv`, or to the

`MPI_Wait`/`MPI_Test` family of functions with a nonblocking receive request, is satisfied by any single message received from a sending replica. Further replica messages, which may support or contradict the original, are processed opportunistically at later times. If the application is confirmed to have proceeded with a correct message, no action needs to be taken (and any remaining messages can be ignored), whereas if the initial message is found to be incorrect, correction can be initiated. To satisfy the final requirement, when the application calls `MPI_Finalize`, the library waits until all messages have been verified.

Naturally, the messages considered in satisfying the first two requirements must correspond to (and, absent corruption, match) the messages that would have been received in synchronized or non-replicated execution. The MPI standard includes a non-overtaking requirement that helps to guarantee this.

> (3.5) Messages are *non-overtaking*: If a sender sends two messages in succession to the same destination, and both match the same receive, then this operation cannot receive the second message if the first one is still pending. If a receiver posts two receives in succession, and both match the same message, then the second receive operation cannot be satisfied by this message, if the first one is still pending.
>
> ...
>
> (3.7.4) Nonblocking communication operations are ordered according to the execution order of the calls that initiate the communication. The non-overtaking requirement of Section 3.5 is extended to nonblocking communication, with this definition of order being used. [3]

Lazy fault detection does not change the order in which communications are initiated at either the sender or receiver. Therefore, by the non-overtaking requirement, it does not affect which messages will satisfy which receive calls. In turn, since repli-

cated sending as in RedMPI sends all replica messages in immediate succession, the ordering of these corresponds directly to the ordering of messages in non-replicated execution.

3.1 Correction

If message corruption can be detected before the message is returned to the application, as in RedMPI, correction is simple: replace the message contents with the correct value. A lazily detected fault, however, is not readily corrected, since the process may have already used the incorrect value and performed further computation and communication. Instead, some form of rollback is required so that the process can proceed as if the correct value was initially received.

One simple approach would be to integrate the SDC detection protocol with a standard checkpoint/restart system, and when a corrupted message is detected, restart the entire application from a point before this message was received. However, a full application restart costs substantial computation time, and this has identical scaling behavior to the checkpointing system alone.

It would also suffice to roll back just the receiving process to the point just before receiving the corrupted message, and replay this and the following messages received up until the point of detection. Given conservative handling of requests, any messages received later than the corrupted one will also be validated later and thus still available in memory for correction. Outgoing messages sent by the process during replay can be discarded; if the original versions of these were tainted by the initial corruption, their receivers will be responsible for performing their own correction if necessary.

Chapter 4

IMPLEMENTATION

We extended the RedMPI library to provide asynchronous methods of SDC detection that avoid waiting for a full set of messages from sending replicas.

As in RedMPI, this functionality is achieved using the MPI profiling layer, enabling compatibility with any existing MPI implementation or application. In addition to exporting `MPI_*` symbols for functions, MPI implementations also export a parallel set of `PMPI_*` symbols that reference the same functions. Our library, like RedMPI, exports new `MPI_*` symbols while using the `PMPI_*` symbols internally.

RedMPI tracks outstanding send and receive requests in an internally managed array. Each request entry references the real underlying MPI requests and, in the case of receive requests, receive buffers for each of these.

4.1 Async Method

Our first extension is an *Async* SDC detection method added to RedMPI. The *Async* method functions similarly to the existing *AllToAll* method: complete messages are sent from each sending replica to each receiving replica, and receiving replicas perform error detection and correction independently of each other.

To support asynchronous operation, we extended the library to also track, when using the *Async* method,

- whether a request entry has returned an initial result to the application (and is therefore *incomplete*)
- which request's buffer was used to return an initial result

```
Persistent state: requests, buffers, appBuf, firstIndex
  for i = 0 to r − 1 do
    completed[i] ← PMPI_Test(requests[i])
  end for
  if firstIndex is unset then
    firstIndex ← smallest i such that completed[i]
    appBuf ← buffers[firstIndex]
  end if
  if all completed then
    if a majority of buffers match buffers[firstIndex] then
      free request information
    else if a majority of buffers match then
      initiate correction using majority value
    else
      abort execution
    end if
  else
    mark request as incomplete
  end if
```

Figure 4.1: Async integrity verification algorithm

- a pointer to the application's buffer, which is independent of any of the underlying receive buffers
- hashes of each received buffer, for more efficient comparisons across multiple rounds of verification

The core algorithmic addition is an integrity-verification routine for the *Async* method, which may be invoked repeatedly on a single request. When provided with an application receive request, the routine operates as outlined in Figure 4.1.

Asynchrony is introduced in the implementation of `MPI_Wait`. The new implementation performs a `PMPI_Waitany` instead of a `PMPI_Waitall` on the underlying requests, and invokes the integrity-verification routine (thus returning the first received buffer immediately to the application).

To fully decouple replicas, send requests must also be made asynchronous; otherwise, a fast process may be blocked waiting for a single slow receiving replica to post

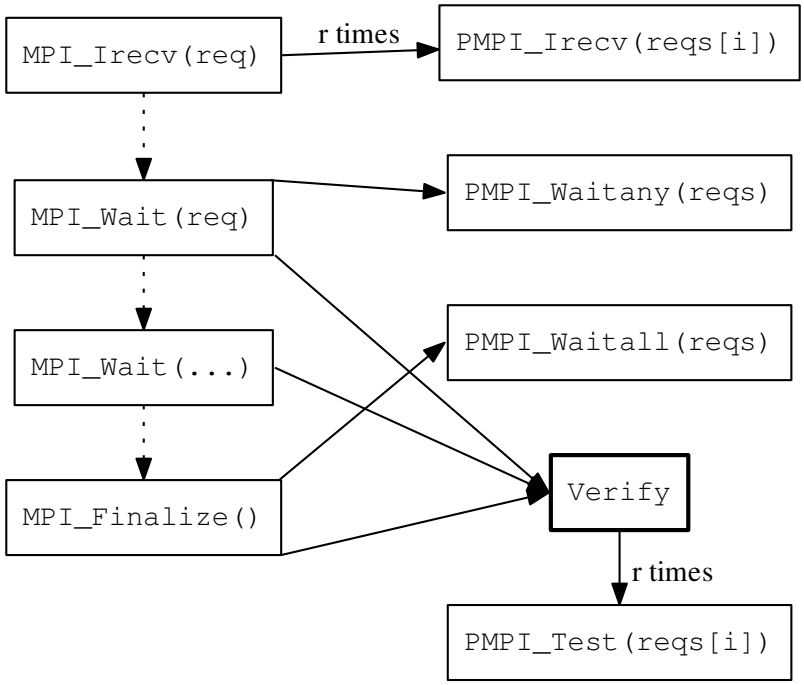

Figure 4.2: Stages of handling a receive request using Async detection

a matching receive call. Since a successful return from `MPI_Wait` for a send request indicates that the application is allowed to reuse the send buffer, this necessitates making an internal copy of each send buffer for the underlying requests to use. When using the *Async* method, `MPI_Isend` makes this copy, and `MPI_Wait` returns immediately for a send request. The integrity-verification routine cleans up such request entries if all underlying requests have completed.

To handle later-arriving replica messages, every execution of `MPI_Wait` first invokes the integrity-verification routine on each *incomplete* request. The execution of `MPI_Finalize` similarly invokes integrity verification, after performing a `PMPI_Waitall` to catch all outstanding messages, on each *incomplete* request. Figure 4.2 gives an overview of the steps that may be taken in processing an application receive request `req` in terms of operations on its underlying requests `reqs`.

Compared to *AllToAll*, *Async* requires a single extra message buffer to be allocated for each request, plus r additional small hash buffers and several more bytes of information. More importantly, these allocations will tend to have greater temporal overlap, since the application may create new requests while others are still in an *incomplete* state. The execution time of the integrity-verification algorithm is minimal, but it may in general be invoked an indeterminate number of times on a single request before successfully completing.

4.2 AsyncHash Method

Our second extension is an *AsyncHash* SDC detection method based on the *MsgPlusHash* method in RedMPI. In *MsgPlusHash*, complete messages are only sent to the replica with the same replica rank k as the sender, and a hash of the message contents is sent to the replica with rank $(k + 1) \bmod r$. Receivers check the hash against the message received and, if necessary, coordinate with adjacent replicas to determine the correct message contents. In *AsyncHash*, the hash message is allowed to be sent and received without blocking the application.

The additional per-request state for the *AsyncHash* method consists of

- an *incomplete* flag, as for *Async*
- a locally-computed hash of the full message received

The integrity-verification routine for the *AsyncHash* method is shown in Figure 4.3. Once the hash message has been received, the routine checks for consistency before cleaning up the request.

Under the *AsyncHash* method, `MPI_Wait` only invokes `PMPI_Wait` to wait for the full message to be sent or received. This is safe since the hash buffers for both sender

```
Persistent state: hashReq, hashBuf, appBuf, localHash
  if localHash is unset then
    localHash ← H(appBuf)
  end if
  completed ← PMPI_Test(hashReq)
  if completed then
    if hashBuf = localHash then
      free request information
    else
      initiate correction
    end if
  else
    mark request as incomplete
  end if
```

Figure 4.3: AsyncHash integrity verification algorithm

and receiver are not exposed directly to the application. Receipt of the hash message is checked for in later calls to `MPI_Wait` and `MPI_Finalize`, as with *Async*.

For SDC correction, *AsyncHash* would require coordination between two replica processes. If corruption occurs at a single sending process, two receiving processes will discover mismatched hashes, but the higher-ranked replica will have a correct message and the lower-ranked replica a matching hash from a distinct sender. These can be used to determine the correct message contents and perform correction at the lower-ranked replica, which will have proceeded in its execution with a corrupted message.

Unlike the *Async* method, *AsyncHash* does not allow the application to complete faster than any single set of processes, since each process is dependent upon the rest of its replica set (those processes sharing the same replica rank) for data. However, it does admit a shadow computing configuration in which some replica sets operate at higher power than others. In this configuration, the faster replica sets will buffer hash messages to be exchanged with the slower.

The additional storage requirements for *AsyncHash* are small and constant, and

the integrity-verification algorithm is very simple, but the same considerations regarding allocation lifespan apply as for *Async*.

Chapter 5

EVALUATION

We evaluated the performance of our modified SDC detection methods relative to the original RedMPI methods and to non-replicated Open MPI 1.6. This measured the performance impact only in the absence of faults, since SDC correction was not implemented. Two different environments were used for evaluation:

1. 32 nodes, each with a 2.3GHz 14-core Intel Xeon processor and 32GB of RAM, running Linux 2.6 and connected via a gigabit Ethernet switch. We launched up to 12 processes per node for effective utilization without interference from system background tasks.

2. 27 nodes[1], each with a 4-core NVIDIA Tegra K1 processor and 2GB of RAM, running Linux 3.10 and connected via a gigabit Ethernet switch. This environment is the Astro cluster designed at Cal Poly [46]. We launched up to 4 processes per node.

The target applications were selected from the NAS Parallel Benchmark suite (NPB) [5]. The MPI implementations of NPB benchmarks use a simple subset of MPI features:

- point-to-point nonblocking communications, including `MPI_Isend`, `MPI_Irecv`, and `MPI_Wait` but not `MPI_Test` or its variants

- blocking communications, which RedMPI implements in terms of nonblocking communications

[1] Although 46 nodes are available in this environment, at most 27 were used in these executions due to benchmark sizing constraints.

- various collective operations, which RedMPI implements in terms of point-to-point communications

The NPB programs exhibit strong scaling: the total amount of work for a given problem size does not depend on the number of processes. Therefore, varying the number of processes changes the communication/computation ratio. In addition, the different benchmarks exhibit a range of communication patterns. The FT benchmark (in Environment 1) and the MG benchmark (in both environments) were excluded due to difficulties encountered when running these under RedMPI.

5.1 Runtime Measurements

We executed each benchmark in each environment, first using only Open MPI, then using RedMPI under double and triple replication and with each of the four SDC detection methods (*AllToAll* and *MsgPlusHash* from RedMPI along with our *Async* and *AsyncHash*). We selected three different process counts for each benchmark, as appropriate for that benchmark's requirements. Benchmark classes (problem sizes) were selected to give approximately a 1–2 minute running time under Open MPI, and in some cases (notably with the IS benchmark) to fit within system memory constraints).

Table 5.1 and Table 5.2 show measurements of runtime for each execution in the two environments. Time was measured across the entire execution of the `mpirun` command rather than using the benchmark code's own timing, which uses `MPI_Wtime` and thus (due to RedMPI's handling of this operation) only represents the elapsed time for the first replica set. Figures 5.1 and 5.2 chart runtimes for double and triple replication respectively, normalized against the runtime under Open MPI, and restricted to Environment 1 and the process count of 64 common to all benchmarks. Figures 5.3 and 5.4 show similar data for Environment 2 and a process count of 16.

Benchmark (Class)	Procs	Open MPI	AllToAll		Async		Improvement		MsgPlusHash		AsyncHash		Improvement	
		1x	2x	3x	2x	3x	2x	3x	2x	3x	2x	3x	2x	3x
BT (B)	36	61.52	215.25	399.67	82.90	117.56	61.5%	70.6%	67.48	67.85	60.50	58.52	10.3%	13.8%
	64	73.87	213.82	359.82	82.49	100.39	61.4%	72.1%	94.36	92.53	72.14	79.85	23.5%	13.7%
	121	80.09	187.13	317.48	83.76	99.33	55.2%	68.7%	97.28	118.70	90.29	104.34	7.2%	12.1%
CG (C)	32	86.42	578.17	968.71	205.40	318.80	64.5%	67.1%	198.54	220.15	146.73	130.29	26.1%	40.8%
	64	172.92	614.23	1078.34	213.96	344.13	65.2%	68.1%	247.40	294.24	174.54	169.72	29.5%	42.3%
	128	295.35	646.69	1157.28	193.90	233.12	70.0%	79.9%	374.68	414.55	285.29	282.79	23.9%	31.8%
EP (D)	32	175.99	177.17	178.62	178.52	179.82	-0.8%	-0.7%	178.48	178.82	180.12	176.64	-0.9%	1.2%
	64	91.01	91.17	90.89	91.32	90.10	-0.2%	0.9%	91.56	90.84	91.28	91.36	0.3%	-0.6%
	128	46.54	48.20	49.15	48.79	48.84	-1.2%	0.6%	48.35	51.95	48.56	55.09	-0.4%	-6.0%
IS (D)	32	263.10	560.24	898.02	541.45	858.65	3.4%	4.4%	297.12	299.30	293.88	289.61	1.1%	3.2%
	64	185.63	366.21	595.17	314.80	491.52	14.0%	17.4%	189.01	268.07	171.37	177.29	9.3%	33.9%
	128	150.14	276.94	445.46	194.10	296.71	29.9%	33.4%	286.60	259.64	140.73	147.41	50.9%	43.2%
LU (B)	32	33.03	91.49	207.56	45.51	65.78	50.3%	68.3%	40.52	49.17	51.90	52.29	-28.1%	-6.3%
	64	26.90	114.72	227.96	43.75	61.68	61.9%	72.9%	67.61	87.09	49.82	54.92	26.3%	36.9%
	128	56.33	139.93	267.97	53.19	54.42	62.0%	79.7%	127.32	152.41	81.20	90.10	36.2%	40.9%
SP (B)	36	117.11	374.56	634.59	143.82	200.20	61.6%	68.5%	120.28	115.75	100.76	99.30	16.2%	14.2%
	64	148.76	404.16	688.51	153.46	182.74	62.0%	73.5%	176.59	174.46	138.23	146.89	21.7%	15.8%
	121	144.59	306.32	495.65	147.35	160.74	51.9%	67.6%	169.78	214.90	160.69	188.91	5.4%	12.1%

Table 5.1: Execution times (seconds) for NPB (Environment 1)

Benchmark (Class)	Procs	Open MPI	AllToAll		Async		Improvement		MsgPlusHash		AsyncHash		Improvement	
		1x	2x	3x	2x	3x	2x	3x	2x	3x	2x	3x	2x	3x
BT (B)	16	101.53	200.01	321.81	132.35	163.10	33.8%	49.3%	118.72	120.64	113.11	113.53	4.7%	5.9%
	36	97.61	222.97	377.83	88.75	117.35	60.2%	68.9%	114.59	129.21	99.96	101.78	12.8%	21.2%
CG (C)	16	163.95	429.28	779.35	299.17	406.45	30.3%	47.8%	199.04	219.87	191.56	191.86	3.8%	12.7%
	32	186.25	478.98	885.79	180.65	253.85	62.3%	71.3%	214.59	255.85	154.78	153.66	27.9%	39.9%
EP (C)	16	41.22	41.45	41.24	41.06	41.58	0.9%	-0.8%	41.06	41.20	41.07	41.19	-0.0%	0.0%
	32	21.07	21.46	21.73	21.24	41.51	1.0%	-91.0%	21.31	21.38	21.27	21.43	0.2%	-0.2%
FT (B)	16	53.47	100.08	155.93	97.66	145.12	2.4%	6.9%	56.01	55.78	55.93	56.19	0.1%	-0.7%
	32	58.90	63.51	96.13	57.73	83.59	9.1%	13.0%	35.46	35.52	35.84	35.62	-1.1%	-0.3%
IS (C)	16	24.50	53.80	82.62	52.34	78.46	2.7%	5.0%	28.81	28.79	29.11	28.59	-1.0%	0.7%
	32	29.24	36.03	57.47	32.22	47.58	10.6%	17.2%	19.36	19.58	18.94	19.82	2.2%	-1.2%
LU (B)	16	99.51	153.07	225.65	211.55	240.66	-38.2%	-6.7%	133.15	139.17	216.30	219.76	-62.4%	-57.9%
	32	54.88	140.33	214.05	153.09	182.45	-9.1%	14.8%	94.03	98.24	187.82	192.84	-99.7%	-96.3%
SP (A)	16	84.22	203.52	323.69	75.24	101.08	63.0%	68.8%	116.84	124.69	86.09	87.60	26.3%	29.7%
	36	97.48	223.61	363.09	58.05	75.27	74.0%	79.3%	120.10	131.23	101.64	105.27	15.4%	19.8%

Table 5.2: Execution times (seconds) for NPB (Environment 2)

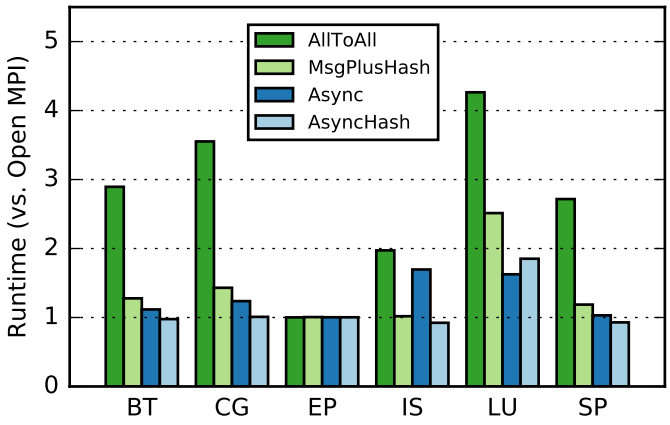

Figure 5.1: Runtime at double replication (64 processes, Environment 1)

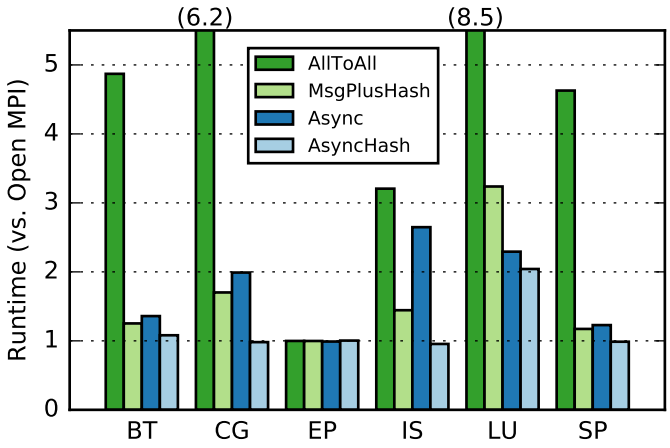

Figure 5.2: Runtime at triple replication (64 processes, Environment 1)

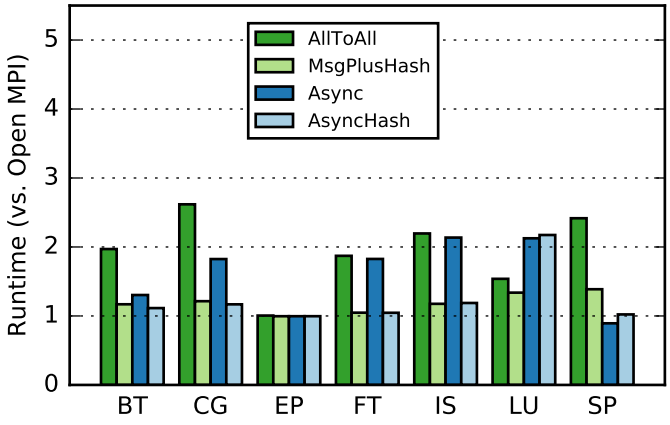

Figure 5.3: Runtime at double replication (16 processes, Environment 2)

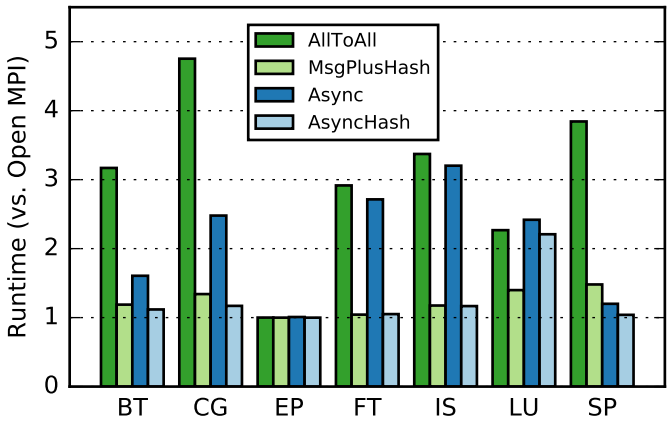

Figure 5.4: Runtime at triple replication (16 processes, Environment 2)

We see that, in our environments, there is a significant penalty for most benchmarks due to the doubled or tripled message traffic with *AllToAll*, and a lesser one for *MsgPlusHash*. In general, our *Async* and *AsyncHash* SDC detection methods alleviate these respective overheads substantially. For most benchmarks, the improvement is consistent across a range of communication/computation ratios, with IS especially sensitive to this ratio and LU exhibiting a few anomalies. The EP benchmark, which performs minimal communication, verifies the reliability of our measurements and shows that there is negligible fixed overhead introduced by SDC detection. For BT, CG, IS, and SP, the runtime using *AsyncHash* is comparable to or slightly better than the baseline Open MPI runtime; the apparent improvements are likely due to different process placement under replication.

5.2 Memory Overhead Measurements

Our *Async* and *AsyncHash* SDC detection methods make a time-memory tradeoff by continuing execution before fully verifying and freeing an application request. To measure the impact on memory usage, we instrumented the allocation of internal request structures in RedMPI across all processes in a replicated job. With this

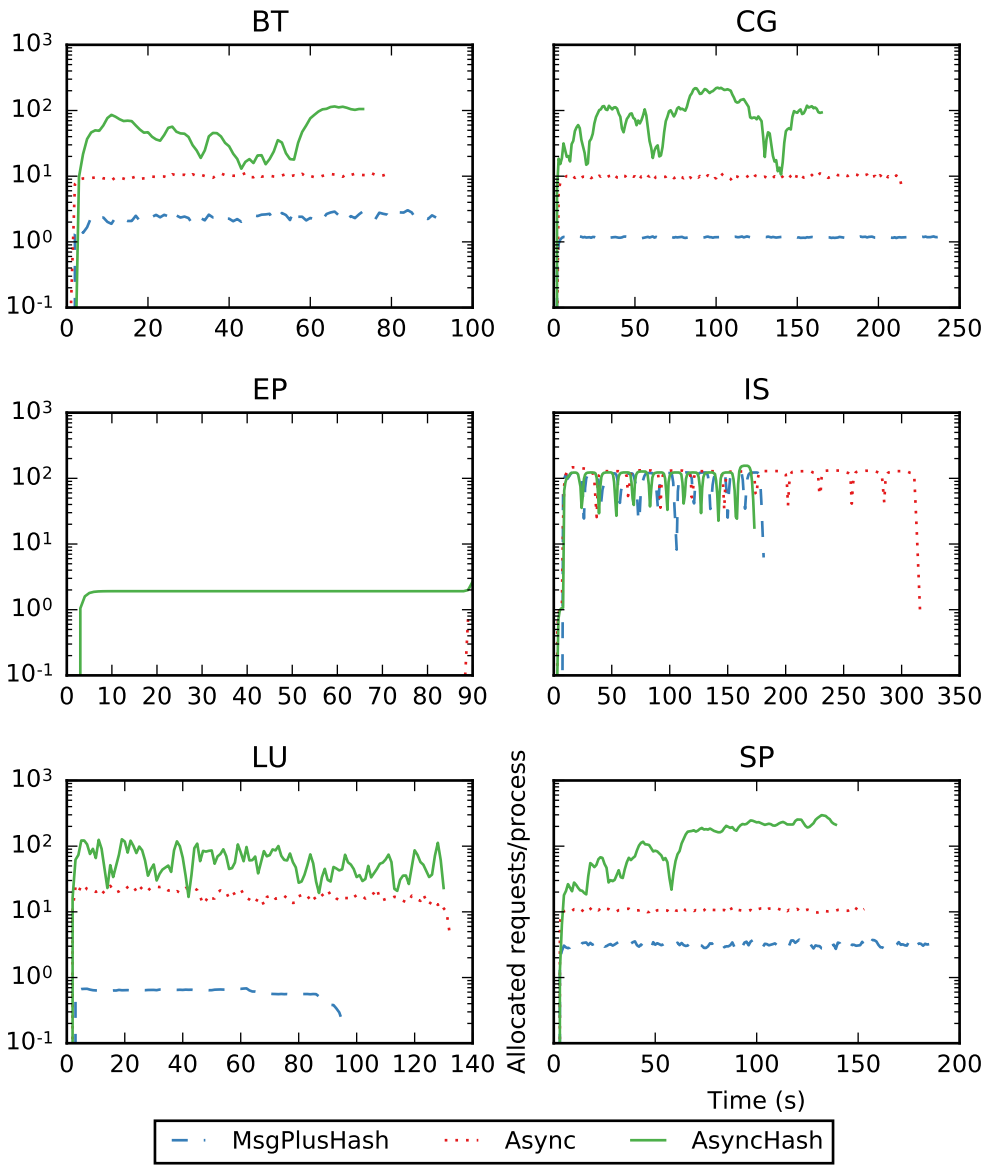

Figure 5.5: Allocated requests per process over time for double-replicated 64-process jobs

instrumentation, each benchmark was executed again in Environment 1 with double replication and a process count of 64 under each of the SDC detection methods.

The resulting data, after applying a moving average, sampling this every second, and normalizing to a per-process count, is plotted for each benchmark in the sub-plots of Figure 5.5. Data for *AllToAll* was excluded since it is nearly identical to that for *MsgPlusHash* but extends for a longer time period in most cases.

Most of the benchmarks appear to frequently block on requests, so under *MsgPlusHash* (or *AllToAll*) there are at most a handful of request entries allocated at a time. As a general trend, *Async* maintains approximately an order of magnitude more requests, while *AsyncHash* shows more dramatic variation over time but has up to 20 times again as many requests allocated. The exceptions are EP, which makes no MPI requests during the majority of its execution (and therefore also does not provide our integrity-verification routine an opportunity to clean up leftover initial requests), and IS, which performs all-to-all communications that appear as 128 individual point-to-point requests and that are not effectively streamlined by our asynchronous methods.

The largest overall memory overhead observed in this data (excluding IS, which has large memory usage regardless of SDC detection) is for the CG benchmark using the *Async* SDC detection method, where an average of 16 requests per process were concurrently allocated at one point in time. A typical message buffer in this benchmark is 150KB, giving a per-process memory overhead of 3.6MB for redundant buffers (one for send requests, two for receive requests) in this situation. With the *AsyncHash* method, request buffers are not copied, so a single allocated request represents less than 200 bytes of memory. The largest number of allocated requests observed in the *AsyncHash* data is 330 per process for the SP benchmark, resulting in approximately 66KB memory overhead. We believe these overheads are inconsequential for current systems and do not present a hindrance to the use of lazy fault detection.

5.3 Unit Testing

Due to the complexity of the *Async* verification routine, it is itself verified with a set of unit tests. These tests cover several different scenarios:

1. All received message buffers are identical.

2. The first two message buffers received are identical (providing confirmation), but a third differs.

3. The first message buffer received is corrupted, but the second and third match (triggering correction).

The tests evaluate the routine's functionality independent of a full MPI environment by providing a mock implementation of the `PMPI_Test` function. All tests pass.

Chapter 6

RELATED WORK

6.1 Replication

6.1.1 Replicated MPI Implementations

Several libraries implementing transparent replication for MPI applications have previously been developed. One early implementation is rMPI, which and replicates MPI processes on multiple nodes to protect against fail-stop failures [48, 19]. MR-MPI is a more sophisticated implementation of redundancy featuring MPI collectives, wildcard receives, and partial replication [17]. PaRep-MPI [23] implements adaptive replication based on failure prediction. All three of these libraries use the MPI profiling layer to transparently interpose between existing MPI implementations and applications.

RedMPI extends the approach of MR-MPI to handle silent data corruption [20]. Its core functionality consists of protocols for MPI point-to-point communication that provide SDC detection. The first method, *AllToAll*, directs (by instrumenting `MPI_Isend`) sent messages to all replicas of the receiving process. The second, *MsgPlusHash*, saves bandwidth by sending a full message to only one receiving replica and a hash to another. The same combination of messages is seen when receiving; `MPI_Irecv` is instrumented to start multiple receive requests, while `MPI_Wait` waits for all these requests to complete and then checks for consistency. Either of these methods provides SDC detection under double replication and SDC correction (by voting) under triple replication.

VolpexMPI [35] and FMI [42] are complete runtime implementations that provide replication, replacing a normal MPI implementation. These approaches consider only

fail-stop failures, so while they do not require close synchronization between replicas, SDC will not be detected.

Purdy et al. [39] propose replicating MPI processes to improve runtime in the face of highly variable latency in a public cloud environment. This work does not include an implementation and does not consider SDC detection.

6.1.2 Process Management

Another body of work within resilience concentrates on management of the replica processes themselves. Shadow computing executes non-primary replica processes at slower speeds [36, 12, 13]. When the primary processes are successful, the replicas can be terminated early, saving power. However, when one of the primary processes fails, a replica can take over and accelerate to full speed, having only a fraction of the work to make up compared to a complete restart. Allowing for SDC detection within shadow computing is one goal of our work.

Node cloning dynamically creates replicas of running processes [40, 41]. This is achieved by copying memory pages to a new node after quiescing MPI communication. Cloning is useful to consolidate divergent node states, or to regain replication after nodes are lost to failures. This complements fault detection by providing a way to recover from even fail-stop failures.

6.1.3 Redundant Multithreading

Redundant multithreading is similar to replication, but in a shared-memory multithreading context. In redundant multithreading, network failures are not a concern, but transient hardware faults are still important [37]. Substantial work by Hukerikar et al. develops redundant multithreading as a fault-detection strategy [30].

Later extensions to this work additionally incorporate lazy fault detection [29, 31]. This form of fault detection minimizes synchronization between replica threads by instead periodically saving state to memory buffers for later comparison. We use this design as inspiration for our MPI implementation of lazy fault detection.

6.2 Other Approaches to Resilience

Beyond checkpointing and replication, introduced in Chapter 2, a wide spectrum of other resilience techniques are possible.

6.2.1 ULFM

Fault-tolerance tools such as BLCR and RedMPI often leave the application unaware of faults and recovery. While this simplifies application development, it requires conservative assumptions to safely protect all applications, and a given application may be able to more efficiently respond to faults itself. User-level failure mitigation (ULFM) is a proposed extension to the MPI standard that exposes fault-tolerance capabilities to the application [6]. ULFM has been evaluated in practical use [7, 22, 34] and consensus algorithms have been developed [26] to allow coordination in recovery.

6.2.2 LFLR

Local failure, local recovery (LFLR) is a fault-tolerance paradigm that addresses these scaling concerns by recovering nodes independently. In LFLR, a single node is maintained as a hot spare containing the parity of every active node's state. An implementation of LFLR has been developed on top of MPI-ULFM [49].

6.2.3 Algorithm-Based Fault Tolerance

Algorithm-based fault tolerance (ABFT) makes use of characteristics of a particular algorithm to detect and mitigate faults. An early case of this applies to matrix operations [28]. ABFT can be used as a starting point to understand other approaches to resilient algorithms [27]

6.2.4 Fault Injection

Simulated faults are useful for testing fault tolerance techniques. Fault injection attempts to mimic the effects of real hardware faults, but at a higher frequency, allowing testing to be performed on smaller systems and timescales. Many fault tolerance implementations are evaluated using some form of fault injection. Cho et al. provide a very good evaluation of various fault injection approaches in terms of the resulting application failures [11]. At a higher level of abstraction, the effect of faults can even be modeled through statistical simulation [51].

Chapter 7

CONCLUSIONS & FUTURE WORK

We have developed an approach to lazy fault detection for MPI programs that improves on previous replication models in efficiency and scheduling flexibility. The evaluation results suggest that our *Async* and *AsyncHash* methods of SDC detection for RedMPI provide a noticeable performance benefit, matching the speed of non-replicated execution for some benchmarks, with reasonable memory overhead. Further, when combined with shadow computing, lazy fault detection can save computational power while still providing SDC detection. Ultimately, we believe this represents a valuable advance in addressing resilience.

The most significant area for future work is implementation of SDC correction (discussed in Section 3.1) for lazy fault detection. With correction added, the efficiency of the library can be evaluated again in the presence of injected faults that corrupt MPI messages.

In addition, further MPI features could be made to avoid replica synchronization, though often with a loss of complete consistency between replicas. RedMPI implements features such as wildcard receives, `MPI_Test`, `MPI_Iprobe`, and `MPI_Wtime` by sharing one replica's result with others. A given application might or might not diverge based on the results of these time-sensitive calls. Adapting these calls to use local results would risk replica divergence for some applications that use them, but improve performance for others.

BIBLIOGRAPHY

[1] MPI: A message-passing interface standard: Version 1.3. Technical report, Message Passing Interface Forum, May 2008.

[2] MPI: A message-passing interface standard: Version 2.2. Technical report, Message Passing Interface Forum, September 2009.

[3] MPI: A message-passing interface standard: Version 3.1. Technical report, Message Passing Interface Forum, June 2015.

[4] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan. 2004.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, Sept. 1991.

[6] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of MPI communication capability Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244–254, Aug. 2013.

[7] W. Bland, A. Bouteiller, T. Herault, J. Hursey, G. Bosilca, and J. J. Dongarra. An evaluation of User-Level Failure Mitigation support in MPI. *Computing*, 95(12):1171–1184, May 2013.

[8] A. Bouteiller. Fault-Tolerant MPI. In T. Herault and Y. Robert, editors, *Fault-Tolerance Techniques for High-Performance Computing*, Computer Communications and Networks, pages 145–228. Springer International Publishing, 2015. DOI: 10.1007/978-3-319-20943-2_3.

[9] J. Brandt, B. Debusschere, A. Gentile, J. Mayo, P. Pebay, D. Thompson, and M. Wong. Using Probabilistic Characterization to Reduce Runtime Faults in HPC Systems. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008. CCGRID '08*, pages 759–764, May 2008.

[10] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward Exascale Resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1):5–28, June 2014.

[11] H. Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–10, May 2013.

[12] X. Cui, B. Mills, T. Znati, and R. Melhem. Shadow Replication: An Energy-Aware, Fault-Tolerant Computational Model for Green Cloud Computing. *Energies*, 7(8):5151–5176, Aug. 2014.

[13] X. Cui, T. Znati, and R. Melhem. Lazy Shadowing: An Adaptive, power-Aware, Resiliency Framework for Extreme Scale Computing. 2014.

[14] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303–312, Feb. 2006.

[15] J. Dongarra, T. Herault, and Y. Robert. Fault Tolerance Techniques for High-Performance Computing. In T. Herault and Y. Robert, editors,

Fault-Tolerance Techniques for High-Performance Computing, Computer Communications and Networks, pages 3–85. Springer International Publishing, 2015. DOI: 10.1007/978-3-319-20943-2_1.

[16] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[17] C. Engelmann and S. Böhm. Redundant Execution of HPC Applications with MR-MPI. ACTAPRESS, 2011.

[18] K. Ferreira, R. Riesen, R. Oldfield, J. Stearley, J. Laros, K. Pedretti, and R. Brightwell. Keeping checkpoint/restart viable for exascale systems. *Sandia National Laboratories, Tech. Rep. SAND2011-6815*, Feb. 2012.

[19] 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. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 44:1–44:12, New York, NY, USA, 2011. ACM.

[20] D. Fiala, F. Mueller, C. Engelmann, R. Riesen, K. Ferreira, and R. Brightwell. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[21] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a

next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[22] M. Gamell, K. Teranishi, M. A. Heroux, J. Mayo, H. Kolla, J. Chen, and M. Parashar. Exploring Failure Recovery for Stencil-based Applications at Extreme Scales. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '15, pages 279–282, New York, NY, USA, 2015. ACM.

[23] C. George and S. Vadhiyar. Fault Tolerance on Large Scale Systems using Adaptive Process Replication. 64(8):2213–2225, Aug. 2015.

[24] T. J. Hacker, F. Romero, and C. D. Carothers. An analysis of clustered failures on large supercomputing systems. *Journal of Parallel and Distributed Computing*, 69(7):652–665, July 2009.

[25] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[26] T. Herault, A. Bouteiller, G. Bosilca, M. Gamell, K. Teranishi, M. Parashar, and J. Dongarra. Practical Scalable Consensus for Pseudo-synchronous Distributed Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, pages 31:1–31:12, New York, NY, USA, 2015. ACM.

[27] M. A. Heroux. Toward Resilient Algorithms and Applications. *arXiv:1402.3809 [cs]*, Feb. 2014. arXiv: 1402.3809.

[28] K.-H. Huang and J. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. C-33(6):518–528, June 1984.

[29] S. Hukerikar, P. Diniz, R. Lucas, and K. Teranishi. Opportunistic application-level fault detection through adaptive redundant multithreading. In *2014 International Conference on High Performance Computing Simulation (HPCS)*, pages 243–250, July 2014.

[30] S. Hukerikar, P. C. Diniz, and R. F. Lucas. A Case for Adaptive Redundancy for HPC Resilience. In D. a. Mey, M. Alexander, P. Bientinesi, M. Cannataro, C. Clauss, A. Costan, G. Kecskemeti, C. Morin, L. Ricci, J. Sahuquillo, M. Schulz, V. Scarano, S. L. Scott, and J. Weidendorfer, editors, *Euro-Par 2013: Parallel Processing Workshops*, number 8374 in Lecture Notes in Computer Science, pages 690–697. Springer Berlin Heidelberg, Aug. 2013. DOI: 10.1007/978-3-642-54420-0_67.

[31] S. Hukerikar, K. Teranishi, P. Diniz, and R. Lucas. An evaluation of lazy fault detection based on Adaptive Redundant Multithreading. In *2014 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, Sept. 2014.

[32] R. Koo and S. Toueg. Checkpointing and Rollback-Recovery for Distributed Systems. SE-13(1):23–31, Jan. 1987.

[33] V. Kumar and A. Agarwal. HT-Ring Paxos: Theory of High Throughput State-Machine Replication for Clustered Data Centers. *arXiv:1507.04086 [cs]*, July 2015. arXiv: 1507.04086.

[34] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, and B. R. de Supinski. Evaluating User-Level Fault Tolerance for MPI Applications. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 57:57–57:62, New York, NY, USA, 2014. ACM.

[35] T. LeBlanc, R. Anand, E. Gabriel, and J. Subhlok. VolpexMPI: An MPI Library for Execution of Parallel Applications on Volatile Nodes. In M. Ropo,

J. Westerholm, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 5759 in Lecture Notes in Computer Science, pages 124–133. Springer Berlin Heidelberg, Sept. 2009. DOI: 10.1007/978-3-642-03770-2_19.

[36] B. Mills, T. Znati, and R. Melhem. Shadow Computing: An energy-aware fault tolerant computing model. In *2014 International Conference on Computing, Networking and Communications (ICNC)*, pages 73–77, Feb. 2014.

[37] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed design and evaluation of redundant multi-threading alternatives. In *29th Annual International Symposium on Computer Architecture, 2002. Proceedings*, pages 99–110, 2002.

[38] A. Oliner and J. Stearley. What Supercomputers Say: A Study of Five System Logs. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2007. DSN '07*, pages 575–584, June 2007.

[39] S. Purdy, P. Hunt, and D. Bindel. Process Replication for HPC Applications on the Cloud. Dec. 2010.

[40] A. Rezaei and F. Mueller. Sustained Resilience via Live Process Cloning. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1498–1507, May 2013.

[41] A. Rezaei and F. Mueller. DINO: Divergent Node Cloning for Sustained Redundancy in HPC. In *IEEE Cluster 2015*, Chicago, IL, Sept. 2015.

[42] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. de Supinski, N. Maruyama, and S. Matsuoka. FMI: Fault Tolerant Messaging Interface for Fast and Transparent Recovery. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1225–1234, May 2014.

[43] F. B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[44] F. B. Schneider. What Good are Models and What Models are Good? pages 17–26. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[45] B. Schroeder and G. A. Gibson. Understanding failures in petascale computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007.

[46] S. Sheen. Astro - a low-cost, low-power cluster for CPU-GPU hybrid computing. Master's thesis, California Polytechnic State University, San Luis Obispo, 2016.

[47] J. Stearley and R. Ballance. A preliminary report on red storm RAS performance. Lugano, Switzerland, 2006.

[48] J. R. Stearley, R. E. Riesen, J. H. Laros III, K. B. Ferreira, K. T. T. Pedretti, R. A. Oldfield, T. Kordenbrock, and R. B. Brightwell. Increasing fault resiliency in a message-passing environment. Technical Report SAND2009-6753, Sandia National Laboratories, 2009.

[49] K. Teranishi and M. A. Heroux. Toward Local Failure Local Recovery Resilience Model Using MPI-ULFM. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 51:51–51:56, New York, NY, USA, 2014. ACM.

[50] J. W. Young. A First Order Approximation to the Optimum Checkpoint Interval. *Commun. ACM*, 17(9):530–531, Sept. 1974.

[51] D. Zhao, D. Zhang, K. Wang, and I. Raicu. Exploring Reliability of Exascale Systems Through Simulations. In *Proceedings of the High Performance*

Computing Symposium, HPC '13, pages 1:1–1:9, San Diego, CA, USA, 2013. Society for Computer Simulation International.

APPENDICES

Appendix A

REPRODUCIBILITY OF PRIOR RESULTS

Our results show a much higher overhead for *MsgPlusHash* as compared to Open MPI, than is presented in the RedMPI publications [20]. The largest overhead presented in these publications is 19.4% for a triple-replicated 512-process CG benchmark. By contrast, our results show 40% or larger overheads for triple-replicated CG. We attempted to understand and reconcile this discrepancy.

The RedMPI evaluation environment is described as follows.

> We deployed RedMPI on a medium sized cluster and utilized up to 96 nodes for benchmarking and testing. Each compute node consists of a 2-way SMPs with AMD Opteron 6128 (Magny-Cours) processors of 8 cores per socket (16 cores per node) with 32 GB RAM per node. Nodes are connected via 1Gbps Ethernet for user interactions and management. MPI transport is provided by a 40Gb/s InfiniBand fat tree interconnect. To maximize the compute capacity of each node, we ran up to 16 processes per node.
>
> When launching RedMPI jobs, we map replica processes so that they do not reside on the same physical nodes. This type of mapping is preferred as a fault on a node will not affect multiple replicas of the same process simultaneously (i.e., due to localized power failures for a whole rack). [20]

A notable difference from our evaluation environments is the use of an InfiniBand

interconnect instead of Ethernet.

A.1 Latency and Bandwidth Tests

We measured latency and bandwidth between hosts in our two environments using the `ping` and `iperf` tools respectively.

In both environments, `iperf` reliably reports transfer rates over 900 MB/s between any two hosts, approaching link capacity and well exceeding bandwidth observed during any benchmark run. Several simultaneous such transfers did not appear to interfere with each other, showing that the aggregate capacity of the central switch in each environment is at least several GB/s.

In Environment 1, `ping` reports round-trip latencies averaging 0.22 ms (standard deviation 0.10 ms). In Environment 2, the values are somewhat worse, with a mean of 0.62 ms and standard deviation of 0.23 ms.

Overall network performance (IP/Ethernet) does not appear to be a likely culprit in our observed messaging overheads. However, this does not rule out issues with MPI implementations' use of TCP for communications over these networks.

A.2 Processes-per-Node Experiments

Investigating large overheads seen with replication led to the observation that the placement of processes on nodes dramatically affects overall runtime in our Environment 1.

We compiled a 32-process class-B CG kernel from NPB. We executed this under each of four different MPI implementations available in our environment (without using RedMPI): Open MPI 1.6, Open MPI 1.10, PGI MPICH, and Intel MPI. For each implementation, only the `-npernode` parameter to `mpirun` was varied, which controls

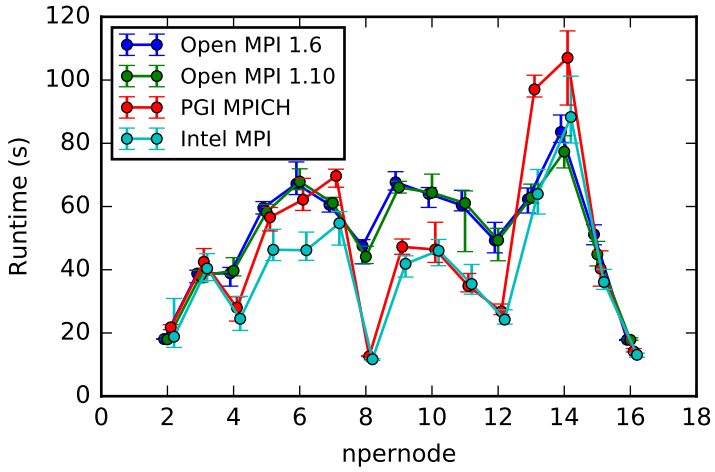

Figure A.1: Runtime by processes-per-node (CG, different implementations)

the number of processes placed on each node before moving to the next. For instance, `mpirun -npernode 10 ... bin/cg.B.32` uses four nodes, placing processes 0–9 on the first, 10–19 on the second, 20–29 on the third, and 30–31 on the fourth.

Figure A.1 shows the runtimes reported by the benchmark code, giving a median, minimum and maximum value across 7 runs for each configuration. There are substantial variations in runtime (some over 300%), and the overall shape is consistent between all four MPI implementations. While some runtime improvements correspond to changes in the number of nodes involved in the execution, there is no clear impact of processor contention once `-npernode` exceeds 14.

We repeated the experiment using Intel MPI and varying the total process count of the benchmark. For larger process counts, `-npernode` values were restricted to those achievable using at most 32 nodes. Figure A.2 shows the results. There is some variation in trends; for instance, `-npernode 12` performs better than 11 or 13 for process counts 64 or less, but worse for process counts of 128 or 256.

We repeated the experiment using the FT benchmark with Intel MPI and varying numbers of processes. Figure A.3 shows the results. For FT, `-npernode` values

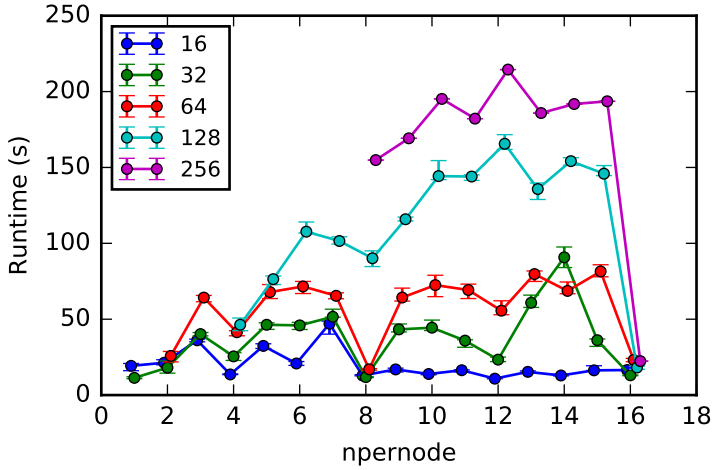

Figure A.2: Runtime by processes-per-node (CG, varying process count)

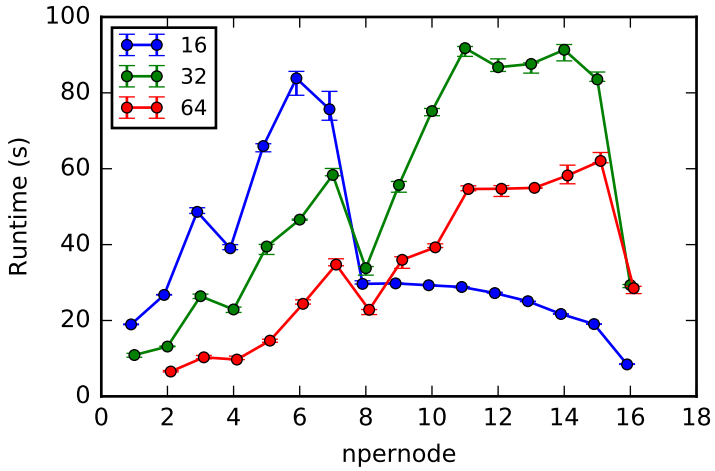

Figure A.3: Runtime by processes-per-node (FT, varying process count)

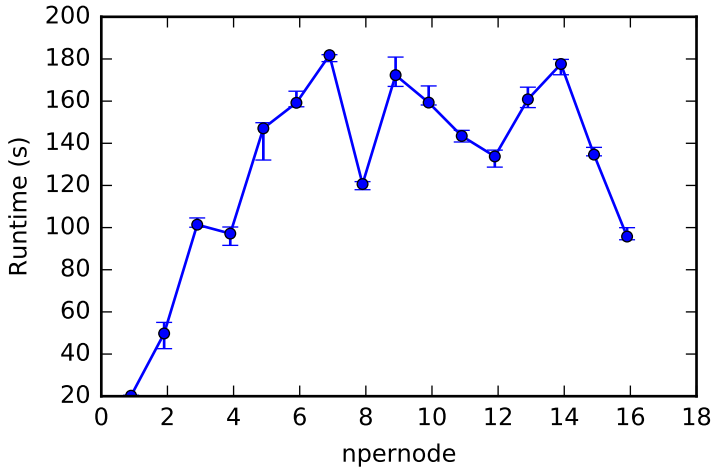

Figure A.4: Runtime by processes-per-node (CG, Environment 2)

between 11 and 15 perform particularly poorly, but powers of 2 are still local minima as for CG.

We repeated the experiment for CG using Open MPI 1.6 in Environment 2. Figure A.4 shows the results. While the absolute values are different, the overall shape is very similar to that for Environment 1.

A.3 Application Profiling

We used profiling tools to introspect a slow execution (of CG with `-npernode 14`). This showed that processes were spending a majority of their time busy-waiting inside `MPI_Wait`.

A.4 Conclusions

Our evaluation environments exhibit significant communication-related runtime variations for NPB that are not intrinsically linked to process replication or to network infrastructure flaws. Further evaluation of our work in an environment with an InfiniBand interconnect, as used for RedMPI validation, would be useful.