Building a Fault Tolerant MPI Application: A Ring Communication Example

Joshua Hursey, Richard L. Graham
Oak Ridge National Laboratory Oak Ridge, TN USA 37831
Email: {hurseyjj,rlgraham}@ornl.gov

Abstract—Process failure is projected to become a normal event for many long running and scalable High Performance Computing (HPC) applications. As such many application developers are investigating Algorithm Based Fault Tolerance (ABFT) techniques to improve the efficiency of application recovery beyond what existing checkpoint/restart techniques alone can provide. Unfortunately for these application developers the libraries that their applications depend upon, like Message Passing Interface (MPI), do not have standardized fault tolerance semantics. This paper introduces the reader to a set of run-through stabilization semantics being developed by the MPI Forum’s Fault Tolerance Working Group to support ABFT. Using a well-known ring communication program as the running example, this paper illustrates to application developers new to ABFT some of the issues that arise when designing a fault tolerant application. The ring program allows the paper to focus on the communication-level issues rather than the data preservation mechanisms covered by existing literature. This paper highlights a common set of issues that application developers must address in their design including program control management, duplicate message detection, termination detection, and testing. The discussion provides application developers new to ABFT with an introduction to both new interfaces becoming available, and a range of design issues that they will likely need to address regardless of their research domain.

Keywords-MPI; Fault Tolerance; Algorithm Based Fault Tolerance; Run-through Stabilization

I. INTRODUCTION

Scientists use High Performance Computing (HPC) systems to help solve complex scientific problems that cannot be solved on more traditional computing systems. As these applications run longer and scale further to address increasingly complex scientific questions, they begin to exceed the reliability of a given HPC system. Administrators of large HPC systems often measure system reliability, in terms of mean time to failure (MTTF), in days or weeks [1]. It is anticipated that future exascale HPC systems will reduce the MTTF to minutes or hours, further exposing the application to the risk of failure during normal computation. Process failures, in particular, will no longer be rare events, but normal events that the application must be prepared to handle [2].

In light of this, applications are looking to augment (or replace) their existing checkpoint/restart fault tolerance techniques with more application focused, Algorithm Based Fault Tolerance (ABFT) techniques to improve the efficiency of application recovery after process failure. Unfortunately, many of the software libraries that HPC applications depend upon are not resilient enough to support ABFT. A library fundamental to many HPC applications is the Message Passing Interface (MPI) [3]. Applications are looking to the MPI standard for a foundation of reliability from which they can develop ABFT techniques. However, the MPI standard does not address how an implementation should behave after a failure, except in the default, abort case (i.e., MPI_ERRORS_ARE_FATAL). So even if the application can find an MPI implementation with reliability semantics, any changes they make to their code will significantly reduce its portability to other HPC systems.

Another daunting challenge for application developers new to the field of ABFT is the amount of both theoretical and practical literature available that they must wade through to understand the range of issues that they will likely encounter during development. Even the practical literature often obscures the communication-level design issues in order to fully describe the high-level algorithmic design and data preservation mechanisms. This can lead an application developer to overlook critical issues such as duplicate message handling and termination detection.

This paper first introduces the reader to a set of run-through stabilization semantics being developed by the MPI Forum’s Fault Tolerance Working Group. Implementations of the proposed interface, like the prototype used in this paper, will allow portable fault tolerant applications and libraries to be built using MPI. Using a well-known ring communication program as the running example, this paper illustrates to application developers some of the issues that arise when creating a fault tolerant variant of their application. We demonstrate how to design a ring program that is able to run-through the failure of multiple processes during normal operation. Process recovery is not addressed in this paper to focus the discussion on the issues that provide the required stabilization functionality.

II. FAULT TOLERANT MPI SEMANTICS AND INTERFACES

The MPI Forum’s Fault Tolerance Working Group is charged with defining a set of semantics and interfaces to enable fault tolerant applications and libraries to be portably constructed on top of the MPI interface. This paper focuses on the run-through stabilization component of the developing proposal which is being extended to include flexible recovery strategies [4]. Run-through stabilization is sufficient for many applications and is a necessary step for applications that may require process recovery. The run-through stabilization component of the proposal provides an application with the ability to continue running and using MPI even when one
The proposal assumes fail-stop process failure meaning that a process is permanently stopped, often due to a crash [5]. For a discussion on how transient failures should be handled by the MPI implementation see the proposal [4]. Other types of faults not currently addressed by the MPI standard (i.e. reliable message delivery), like Byzantine failures [6], are left to the application to address, as necessary.

For our discussion, we assume that the MPI implementation provides the application with a view of the failure detector that is both strongly accurate and strongly complete, thus a perfect failure detector [7]. This means that eventually every failed process will be known to all processes in the MPI universe (strong completeness), and that no process is reported as failed before it actually fails (strong accuracy). The application is notified of a process failure once it attempts to communicate directly (e.g. point-to-point operations) or indirectly (e.g., collective operations) with the failed process through the return code of the function, and error handler set on the associated communicator. This proposal does not change the default error handler of MPI_ERRORS_ARE_FATAL, so the application must explicitly change the error handler to, at least, MPI_ERRORS_RETURN on all communicators involved in fault handling in the application.

The subset of the new interfaces that relate to our discussion are presented in Figure 1. The proposal and corresponding prototype implementation in Open MPI [8] used in this paper currently support all of MPI-1 functionality including collective and group management operations. We are currently extending both the proposal and prototype to support the remainder of the MPI standard including parallel I/O and one-sided operations.

The proposal is based on the principle that the application should explicitly recognize process failures that affect them in each communicator they intend to continue using. Unrecognized process failures continue to throw errors when the failed process is referenced, while recognized process failures have MPI_PROC_NULL semantics and do not throw errors when referenced.

A process can locally query for the state of an individual rank using the MPI_Comm_validate_rank function, or access an array of all failed ranks using the MPI_Comm_validate function. A process can recognize a set of rank failures locally on a specific communicator using the MPI_Comm_validate_clear function. Local recognition of the rank failure allows for the continued use of point-to-point operations with the specified ranks, but not collective operations. Additionally, a process can collectively recognize all failures in a communicator by using the MPI_Comm_validate_all function. The collective validate function returns the total number of failures in that communicator as agreed upon by all of the alive processes in the communicator, and re-enables collective operations on that communicator. This function will return either success everywhere or some error at each alive rank. This means that the MPI_Comm_validate_all function provides the application with an implementation of a fault tolerant consensus algorithm [9]. Failures are recognized on a per-communicator basis to guarantee that libraries are able to receive notification of the failure, even if the main application has previously recognized the failure on a duplicate communicator.

The MPI_Rank_info object is used by the validate functions to express the rank, generation, and state of a specific process. The rank field indicates the rank in the associated communicator. The generation field is a monotonically increasing number that is used to distinguish between multiple recovered versions of a process. Since we are only concerned with run-through stabilization in the paper, this field will not be used. The state field indicates which one of the three states that the rank is in. The MPI_RANK_OK state indicates that the rank is running normally. The MPI_RANK_FAILED state indicates that the rank has failed, and not yet been recognized by this process on this communicator. The MPI_RANK_NULL state indicates that the rank has failed, and has been recognized by this process on this communicator.

As previously mentioned, the application is notified of a process failure once it attempts to communicate directly or indirectly with the failed process. Direct point-to-point communication with non-failed ranks behaves normally even if there are unrecognized process failures in the communicator. If a rank tries to communicate directly with an unrecognized failed rank then the function will return an error in the class MPI_ERR_RANK_FAIL_STOP. If a rank posts a receive to MPI_ANY_SOURCE (an indirect communication) and there is an unrecognized failed rank then the function will return an error in the class MPI_ERR_RANK_FAIL_STOP.

Once any rank fails in a communicator, all collective operations will return an error in the class MPI_ERR_RANK_FAIL_STOP until the communicator is

---

Fig. 1: MPI Communicator Management Extensions

```c
1  MPI_Rank_info {
2    rank, /* Rank in the communicator */
3    generation, /* Generation of this rank */
4    state {
5      MPI_RANK_OK, /* Normal running state */
6      MPI_RANK_FAILED, /* Failed, not recognized */
7      MPI_RANK_NULL /* Recognized as failed */
8    }
9  }
10  /* Local operations */
11  MPI_Comm_validate_rank(comm, rank, rank_info);
12  MPI_Comm_validate(comm, incount, outcount,
13      rank_infos[]);
14  MPI_Comm_validate_clear(comm, count,
15      rank_infos[]);
16  /* Collective operation */
17  MPI_Comm_validate_all(comm, outcount);
18  MPI_Icomm_validate_all(comm, outcount, request);
```
Fig. 2: Traditional fault unaware ring application.

repaired using the collective MPI_Comm_validate_all function. This requirement allows the MPI implementation an opportunity to re-optimize collective operations for improved performance after the failure. Once the communicator has been collectively validated, then recognized failed ranks participate as if they were MPI_PROC_NULL (see [4] for more details). In order to preserve failure-free performance of collective operations, the working group decided to not require consistent return codes from collective operations (with the exception of MPI_Comm_validate_all). For example, if the MPI implementation uses a tree implementation for MPI_Bcast then it is possible for a process to successfully leave the collective early once it has propagated the message to its children. However, a failure may occur while traversing the remainder of the tree that would cause some processes to return error. The MPI_Comm_validate_all function is useful in creating recovery blocks for sets of collective operations [10].

III. NEIGHBOR BASED COMMUNICATION: RING

Usually the first point-to-point MPI program that a student creates is a ring program. This program receives a message from the left rank and sends it to the right rank usually changing the buffer slightly before sending it along. Figure 2 presents pseudo code for such a program, which is also used for some latency benchmarks. The ring application example allows us to focus the reader on the communication-level design issues rather than issues related to, for example, data preservation, as in other ABFT literature mentioned in Section IV.

Figure 3 presents a modified version of the original code which hides some of the fault tolerance complexity in the supporting functions. The first change on Line 10 is to replace the default error handler of MPI_ERRORS_ARE_FATAL with MPI_ERRORS_RETURN for MPI_COMM_WORLD.

Next, the application needs to determine the left and right neighbors of a given process (named \( P_L \) and \( P_R \), respectively for rank \( P \)). The previous calculation for the left and right neighbors (seen in Figure 2 on Lines 9-10) must be checked to ensure they return active ranks. This check prevents the application from interacting with a rank that is already known to be failed, thus wasting effort. Figure 4 presents the new neighbor calculation functions that uses the local MPI_Comm_validate_rank function to skip known failed ranks in the communicator. For this example, we assume that the root process (\( P_{Root} \)) does not fail, and the get_current_root function always returns 0. Removing this limitation is discussed in Section III-D.

Next, we turn our attention to the FT_Send_right and FT_Recv_left functions. In the FT_Send_right func-
int to_left_of(int n) {
    MPI_Rank_info rs;
    do {
        n = (0 == n ? size - 1 : n - 1);
        MPI_Comm_validate_rank(MCW, n, rs);
    } while( MPI_RANK_OK != rs.state );
    if( me == n ) { MPI_Abort(MCW, -1); }
    return n;
}

int to_right_of(int n) {
    MPI_Rank_info rs;
    do {
        n = (n + 1)%size;
        MPI_Comm_validate_rank(MCW, n, rs);
    } while( MPI_RANK_OK != rs.state );
    if( me == n ) { MPI_Abort(MCW, -1); }
    return n;
}

Fig. 4: Fault aware right and left neighbor selection.

int FT_Send_right(int msg_t buffer) {
    failed = false;
    if( MPI_SUCCESS != MPI_Send(buffer, T_N, P_R) ) {
        failed = true;
        P_R = to_right_of(P_R);
    }
    return MPI_SUCCESS;
}

Fig. 5: Fault tolerant send to right neighbor.

A first attempt at the code for the FT_Recv_left might mirror the technique used for FT_Send_right. The FT_Recv_left function would attempt to receive from P_L and, upon failure, search for the next neighbor and resend the receive. This version of the function may seem correct, but consider the scenario seen in Figure 6 where P_1 sends to P_2 and P_2 fails after receiving the buffer but before sending the buffer onto P_3. P_3 will re-post the receive to P_1, but P_1 is already waiting for the next iteration of the ring and does not yet notice the failure of P_2. The result is that the parallel program hangs waiting for progress in the ring that will never occur because the control was lost with P_2. The question is how do we get P_1 to notice P_2 failed in order to resend the buffer to P_3, while still waiting for the next buffer from P_0?

A. Using MPI_Irecv as a Failure Detector

To solve the problem with the FT_Recv_left function illustrated by Figure 6, we take advantage of the new semantics of the MPI receive operation. If a peer fails then all posted MPI receive operations involving that peer will return an error in the class MPI_ERR_RANK_FAIL_STOP. So we can use this semantic and MPI_Irecv to detect if the right peer fails even while waiting for the next ring buffer from the left peer.

Figure 9 presents a version of the FT_Recv_left function that uses an MPI_Irecv posted to the P_R as a fault detection mechanism. Since P_R will never send a message backwards in the ring, the only time this request will complete is if P_R fails. If we determine that P_R fails then we find the next, right neighbor and resend the last buffer sent. If P_L fails then we just repost to the next, left neighbor and wait for it to resend the last buffer, as seen in Figure 7.

Without Lines 24-28 in Figure 9, there is a problem with this version of FT_Recv_left. As illustrated in Figure 8, it is possible that the resend will trigger duplicate messages in the ring. In this example, P_1 sends to P_2, which then sends to P_3. P_2 fails as P_3 sends to P_0. P_1 notices the failure of P_2 and resends the buffer to P_3. P_3 already forwarded on the original buffer when it receives a resent buffer. Since both the normal and resent buffers arrived on the same tag, P_3 is unable
to distinguish them and forwards the resent buffer incorrectly thinking it is from the next iteration of the ring. Duplicate messages like this would lead to multiple completions of the same ring iteration.

B. Controlling for Duplicate Messages

There are a couple of ways to address the problem illustrated by Figure 8 regarding duplicate messages. We could use a separate tag for the resent communication and post two receives to \( P_L \) (or one receive using \( MPI_{\text{ANY\_TAG}} \)). By using a different tag for normal messages and resending, we create two different communication contexts, so messages between these two contexts may be received out of order. For our ring example, this does not impact correctness, but it may for other applications using neighbor based communication.

As an alternative, we can use the same communication context (i.e., same tag, communicator, and peer) and piggyback an iteration marker on the buffer to allow us to detect and drop duplicate, already processed messages. The iteration marker would indicate the current ring iteration, seen in Figure 3 on Lines 17 and 25.

Line 17 of Figure 3 adds the ring iteration marker to the buffer before being transferred among the ranks. A non-root process will increment the iteration marker after it passes along the buffer, seen on Line 25. This will allow it to distinguish between resent and normal buffers when it waits in the modified \( FT_{\text{Recv\_left}} \) function seen in Figure 9 on Lines 24-28. This isolates each iteration as a context of communication.

Figure 10 illustrates how this receive variant avoids duplicate message transmission. Upon a successful receive from \( P_L \), the process checks the iteration marker field of the buffer. If the iteration marker is less than the current generation, then this is a resent message from the last ring iteration and this process has already passed the buffer onto \( P_R \) and can disregard this buffer. If the iteration marker is equal to the current iteration, then this is a resent message for the current ring iteration and this process will need to forward it along to \( P_R \) as normal. If the iteration marker is greater than the current iteration, then this message has been received out of order which will never happen. This will never happen since, as seen in Figure 10, \( P_3 \) would have had to pass control to \( P_0 \) in order for \( P_1 \) to send it a future iteration (unless \( P_1 \) is Byzantine faulty, which is a failure mode not addressed here).

C. Termination Detection

The current ring program is able to run-through multiple, non-root process failures by recovering the ring topology in a local manner. There is one final issue to address, termination detection. In a failure free program all ranks know when the ring operation is finished by counting locally how many times they have participated in the ring operation. Once this number has reached a predefined limit all ranks rendezvous in \( MPI_{\text{Finalize}} \).

In a fault tolerant ring program once a process finishes propagating the last iteration of the ring, it must still stick around to make sure that the ring finishes by resending the buffer as necessary. So how does the algorithm tell all processes that it is time to stop watching their \( P_R \) neighbor and call \( MPI_{\text{Finalize}} \)?
One may think to use MPI_Barrier to determine when all processes have arrived at the end of the program. However, this is not sufficient for two reasons. First, MPI_Barrier is a blocking operation so an MPI_Barrier (scheduled to be included in the MPI 3.0 standard) would need to be used in order to progress the resend messages to $P_R$. Secondly, the return value from the barrier operation is not guaranteed to be consistent across all processes. So some processes may receive success and others an error if a process fails during the barrier operation. It is be possible to use multiple calls to MPI_Barrier to determine if all processes entered the first barrier by inspecting combinations of return codes, but this comes at considerable cost in both performance and complexity of the application program.

If we assume that the root process ($P_{Root}$) cannot fail, then we can have $P_{Root}$ broadcast out a special termination message to all alive processes. Concurrently all non-root processes will be waiting in a receive from $P_{Root}$ (for termination) and $P_R$ (for resending). If $P_{Root}$ fails, then the remaining alive processes will call MPI_Abort since root failure is not supported. Figure 11 presents the pseudo code for this type of termination detection (called in Figure 3 Line 28).

**D. What if the root fails?**

Up to this point we have assumed that $P_{Root}$ does not fail. So what would happen to our algorithm if the root process fails during either the main ring portion or the termination detection portion of the application?

First a new $P_{Root}$ must be chosen by all alive processes. Figure 12 presents a simple leader election algorithm that determines the new root by choosing the lowest rank among all the alive processes in the communicator. For the termination detection function instead of aborting the application when the root fails, a new root should be chosen and will resume broadcasting the termination message. However, such reliable broadcast algorithms are delicate to implement, especially when attempting to improve the scalability of the algorithm [11], [12], [13], [14].

So instead of incorporating the complexity of a reliable broadcast algorithm into our application, we can use the fault tolerant consensus algorithm provided by the MPI implementation (i.e., MPI_Comm_validate_all). Since we still need to progress the ring, we must use the non-blocking form of the function, MPI_Icomm_validate_all. Figure 13 presents the new termination detection pseudo code.

For the main ring portion of the program, once a rank
determines that it has become the root it must regain control
over the loop iteration based upon its current knowledge of
the ring state. The PL peer will resend to the new root the
last buffer it passed to the old root before it failed. From
this information and local knowledge of the last buffer that
it passed to PR, the new root can determine the last known
iteration of the ring. Once it has determined the state of the
ring, it can resume control over the iterations and lead the
remaining processes to completion.

E. Testing

Process failure can occur at any time during application
execution. This paper discussed how to handle various process
failure scenarios that were discovered by code inspection and
fault injection testing using the prototype implementation.
Fault injection is currently the most popular technique avail-
able to application developers [15], [16], [17]. Fault injection
tools allow an application developer to inject failures into their
application during normal execution to test if the application
behaves according to design. Intensive use of fault injection
tools can allow a developer to build confidence in their
solution.

But how can a developer know when they have addressed
all of the problematic fault scenarios in their application? The
debugging, verification, and validation research communities
do not currently have many tools to support MPI application
developers. The lack of support is most likely attributed to the
lack of standardized MPI process fault tolerance semantics
to test applications against. Once MPI provides standardized
process fault tolerance semantics then the various tool de-
veloper communities can start developing tools and adapting
techniques to assist application developers in answering this
critical question.

IV. RELATED WORK

In [18], Gropp and Lusk described how a manager/worker
style MPI program might recover from process loss by using
multiple intercommunicators and forgetting about intercom-
unicators connecting to lost processes. Though they demon-
strated how an application might use a high-quality MPI
implementation to achieve some fault tolerance semantics, this
behavior is not standardized and therefore not portable. This
has been and continues to be a significant barrier for appli-
cation developers that need fault tolerance semantics since
they can only design for a single version of a particular MPI
implementation on a particular HPC machine. Additionally,
the management of multiple sets of intercommunicators for
a single group of processes is cumbersome in comparison
to directly using intracommunicators, as in the run-through
stabilization proposal.

When it comes to extending the MPI standard the FT-
MPI project is the most closely related in terms of semantics
to the run-through stabilization proposal used in this paper.
FT-MPI is an MPI-1 implementation that extended the MPI
communicator states and modified the MPI communicator
construction functions [19]. Fault tolerant MPI applications
use these extensions to stabilize MPI communicators and, op-
tionally, recover failed processes by relaunching them from the
original binary and rejoining them into the MPI communicator.
The run-through stabilization proposal behaves similar to FT-
MPI’s blank communicator mode, where failed processes
are replaced by MPI_PROC_NULL. Additionally, the two
proposals have complementary semantics regarding point-to-
point and collective operations. The main difference between
these projects is in the handling of communicator and group
objects. Upon process failure, FT-MPI destroys all MPI objects
with non-local information (e.g., communicators and groups),
except MPI_COMM_WORLD, requiring the application to
manually recreate these objects after every failure in the
same order. In contrast, the run-through stabilization proposal
preserves all communicators and groups. Additionally, FT-MPI
required that every process failure be recognized globally by
all alive processes in order to complete the recovery stage.
In the run-through stabilization proposal process failures can
be recognized locally, and on a per-communicator basis. These
two differences allow the run-through stabilization proposal to
more flexibly support libraries, and, by allowing for localized
failure recognition, open the door to more scalable fault
tolerance solutions. However, the run-through stabilization
proposal does not, at the moment, handle process recovery
and rejoining recovered processes to existing communicators.

Applications have already started to experiment with in-
tegrating fault tolerance techniques into their code. ABFT
techniques require specialized algorithms that are able to adapt

Fig. 13: Fault tolerant termination detection function using
MPI_Icomm_validate_all

```c
int FT_Termination(buffer) {
    /* For termination agreement */
    MPI_Icomm_validate_all(MCW, cnt, req[Idx_N]);
    /* Use MPI to detect when we need to resend */
    MPI_Irecv(dummy_buff, PR, TN, req[Idx_F]);
    do {
        failed = false;
        ret = MPI_Waitany(2, req, &idx, &status);
        if( MPI_SUCCESS != ret ) {
            failed = true;
            if( idx == Idx_F ) {
                /* Our right peer failed, resend message */
                PR = to_right_of(PR);
                FT_Send_right(buffer);
                MPI_Irecv(dummy_buff, PR, TN, req[Idx_F]);
            } else {
                /* Validate should not fail, but if it does repost */
                MPI_Icomm_validate_all(MCW, cnt, req[Idx_N]);
            }
        }
    } while( failed );
    return MPI_SUCCESS
}
```
to and recover from process loss [20]. ABFT techniques typically require data encoding, algorithm redesign, and diskless checkpointing [21] in addition to a fault tolerant message passing environment (e.g., MPI). Although matrix operations have been the focus of much of the research into ABFT [22], [23], [24], there has also been research in other domains such as heat transfer applications [25].

Related to ABFT is natural fault tolerance techniques. Natural fault tolerance techniques focus on algorithms that can withstand the loss of a process and still get an approximately correct answer, usually without the use of data encoding or checkpointing. So natural fault tolerance can be viewed as a more general form of ABFT [26], [27].

V. CONCLUSION

In future HPC systems process failure is projected to be a normal event that the application must be prepared to handle [2]. In light of this projection, HPC application developers are starting to consider ABFT techniques to improve the efficiency of application recovery after process failure. MPI supports many HPC applications, but lacks standardized process fault tolerance semantics.

This paper introduced the reader to some of the run-through stabilization semantics being developed by the MPI Forum’s Fault Tolerance Working Group. Using the proposed semantics and a well-known ring communication program as a running example, this paper illustrated to application developers new to ABFT some of the issues that arise when creating a fault tolerant variant of their application. The ring application example allowed the discussion to focus on the communication-level issues rather than the data preservation issues covered by existing literature. We highlighted a common set of issues that application developers will need to address regardless of their domain including program control management, duplication message handing, termination detection, and testing.

This paper presented pseudocode for the fault tolerant ring MPI application. The full source code can be found at the link below:


ACKNOWLEDGMENTS

Research sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

REFERENCES