A Proposal for User-Level Failure Mitigation in the MPI-3 Standard

Wesley Bland George Bosilca Aurelien Bouteiller
Thomas Herault
Jack Dongarra
{bland, bosilca, bouteill, herault, dongarra } @ eecs.utk.edu
Innovative Computing Laboratory,
Department of Electrical Engineering and Computer Science,
University of Tennessee, Knoxville

February 24, 2012

Abstract

This chapter describes a flexible approach, providing process fault tolerance by allowing the application to react to failures, while maintaining a minimal execution path in failure-free executions. The focus is on returning control to the application by avoiding deadlocks due to failures within the MPI library. No implicit, asynchronous error notification is required. Instead, functions are provided to allow processes to invalidate any communication object, thus preventing any process from waiting indefinitely on calls involving the invalidated objects. We consider the proposed set of functions to constitute a minimal basis, which allows libraries and applications to increase the fault tolerance capabilities by supporting additional types of failures, and build other desired strategies and consistency models to tolerate faults.

1.1 Introduction

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

The expected behavior of MPI in case of a process failure is defined by the following statements: any MPI call that involves a failed process must not block indefinitely, but either succeed or raise an MPI error (see Section 1.2); An MPI call that does not involve the failed process will complete normally,

unless interrupted by the user through provided functions. Asynchronous failure propagation is not required by the MPI standard. If an application needs global knowledge of failures, it can use the interfaces defined in Section 1.3 to explicitly propagate locally detected failures.

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

1.2 Failure Notification

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

1. the operation is a collective call and the process appears in one of the groups on which the operation is applied;
2. the process is a named or matched destination or source in a point-to-point communication;
3. the operation is an MPI_ANY_SOURCE receive operation and the failed process belongs to the source group.

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

Advice to implementers. It is a legitimate implementation to provide failure detection only for processes involved in an ongoing operation, and postpone detection of other failures until necessary. Moreover, as long as an implementation can complete operations, it may choose to delay returning an error. Another valid implementation might choose to return an error to the user as quickly as possible. (End of advice to implementers.)

Note for the Forum. The text of Page 65, lines 28-33, must be changed to allow MPI_IPROBE to set flag=true and return the appropriate status if an error is detected during an MPI_IPROBE. MPI_PROBE is defined as behaving as MPI_IPROBE so it should be sufficient. Similarly, the same effort should be done for MPI_MPROBE and MPI_MRECV.

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

1.2.1 Point-to-Point and Collective Communication

When a failure prevents the MPI implementation from successfully completing a point-to-point communication, the communication is marked as completed with an error of class MPI_ERR_PROC_FAILED. Further point-to-point communication with the same process on this communicator must also return MPI_ERR_PROC_FAILED.

MPI libraries can not determine if the completion of an unmatched reception operation of type MPI_ANY_SOURCE can succeed when one of the potential senders has failed. If the operation has matched, it is handled a named receive. If the operation has not yet matched and worked on a request allocated by a nonblocking communication call, then the request is still valid and pending and it is marked with an error of class MPI_ERR_PENDING. In all other cases the operation must return MPI_ERR_PROC_FAILED. To acknowledge a failure and discover which processes failed, the user should call MPI_COMM_FAILURE_ACK.

Advice to users. It should be noted that a nonblocking receive from MPI_ANY_SOURCE could return one of three error codes due to process failure. MPI_SUCCESS indicates no failure. MPI_ERR_PROC_FAILED indicates the request has been internally matched and cannot be recovered. MPI_ERR_PENDING indicates that while a process has failed, the request is still pending and can be continued. (End of advice to users.)

When a collective operation cannot be completed because of the failure of an involved process, the collective operation eventually returns an error of class MPI_ERR_PROC_FAILED. The content of the output buffers is undefined.

Advice to users. Depending on how the collective operation is implemented and when a process failure occurs, some participating alive processes may raise an error while other processes return successfully from the same collective operation. For example, in MPI_Bcast, the root process is likely to succeed before a failed process disrupts the operation, resulting in some other processes returning an error. However, it is noteworthy that for non-rooted collective operations on an intracommunicator, processes which do not enter the operation due to failure provoke all surviving ranks to return MPI_ERR_PROC_FAILED. Similarly, on an intercommunicator, processes of the remote group failing before entering the operation have the same effect on all surviving ranks of the local group. (End of advice to users.)

Advice to users. Note that communicator creation functions (like MPI_COMM_DUP or MPI_COMM_SPLIT) are collective operations. As such, if a failure happened during the call, an error might be returned to some processes while others succeed and obtain a new communicator. While it is valid to communicate between processes which succeeded to create the new communicator, it is the responsibility of the user to ensure that all involved processes have a consistent view of the communicator creation, if needed.

A conservative solution is to invalidate (see Section **??** the parent communicator if the operation fails, otherwise call an MPI_Barrier on the parent communicator and invalidate the new communicator if the MPI_Barrier fails. (End of advice to users.)

1.2.2 Dynamic Process Management

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

1. If the MPI implementation decides to return an error related to process failure at the root process of MPI_COMM_CONNECT or MPI_COMM_ACCEPT, the root processes of both intracommunicators must return an error of class MPI_ERR_PROC_FAILED (unless required to return MPI_ERR_INVALIDATED as defined by 1.3.1).

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

Advice to users. As with communicator creation functions, it is possible that if a failure happens during dynamic process management calls, an error might be returned to some processes while others succeed and obtain a new communicator. (End of advice to users.)

1.2.3 One-Sided Communication

As with all non-blocking operations, one-sided communication operations should delay all failure notification to their synchronization calls and return MPI_ERR_PROC_FAILED (see Section 1.2). If the implementation decides to return an error related to process failure from the synchronization function, the epoch behavior is unchanged from the definitions in Section 11.4. Similar to collective operations over MPI communicators, it is possible that some processes could have detected the failure and returned MPI_ERR_PROC_FAILED, while others could have returned MPI_SUCCESS.

Unless specified below, MPI makes no guarantee about the state of memory targeted by any process in an epoch in which operations completed with an error related to process failure.

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

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

operations on the lock must fail with an error of class MPI_ERR_PROC_-FAILED.

Advice to users. It is possible that request based RMA operations complete successfully while the enclosing epoch completes in error due to process failure. In this scenario, the local buffer is valid but the remote targeted memory is undefined. (End of advice to users.)

1.2.4 I/O

Due to the fact that MPI I/O writing operations can choose to buffer data to improve performance, for the purposes of process fault tolerance, all I/O data writing operations are treated as operations which synchronize on MPI_FILE_-SYNC. Therefore (as described for non-blocking operations in Section 1.2), failures may not be reported during an MPI_FILE_WRITE_XXX operation, but must be reported by the next MPI_FILE_SYNC. In this case, all alive processes must uniformly return either success or a failure of class MPI_ERR_PROC_-FAILED.

Once MPI has returned an error of class MPI_ERR_PROC_FAILED, it makes no guarantees about the position of the file pointer following any previous operations. The only way to know the current location is by calling the local functions MPI_FILE_GET_POSITION or MPI_FILE_GET_POSITION_SHARED.

1.3 Failure Mitigation Functions

1.3.1 Communicator Functions

MPI provides no guarantee of global knowledge of a process failure. Only processes involved in a communication with the failed process are guaranteed to eventually detect its failure (see Section 1.2). If global knowledge is required, MPI provides a function to globally invalidate a communicator.

MPI_COMM_INVALIDATE(comm)

IN	comm	**communicator (handle)**

This function notifies all processes in the groups (local and remote) associated with the communicator *comm* that this communicator is now considered invalid. This function is not collective. All alive processes belonging to *comm* will be notified of the invalidation despite failures. An invalid communicator preempts any non-local MPI calls on *comm*, with the exception of MPI_COMM_SHRINK and MPI_COMM_AGREEMENT (and its nonblocking equivalent). A communicator becomes invalid as soon as:

1. MPI_COMM_INVALIDATE is locally called on it;
2. Or any MPI function raised an error of class MPI_ERR_INVALIDATED because another process in *comm* has called MPI_COMM_INVALIDATE.

(or such error field should have been set in the status pertaining to a request on this communicator).

Once a communicator has been invalidated, all subsequent non-local calls on that communicator, with the exception of MPI_COMM_SHRINK and MPI_COMM_AGREEMENT (and its nonblocking equivalent), are considered local and must return with an error of class MPI_ERR_INVALIDATED. If an implementation chooses to implement MPI_COMM_FREE as a local operation (see Page 209 Line 1), it is allowed to succeed on an invalidated communicator.

Note for the Forum. The text of Page 208 lines 39-43 must be amended to provide the following advice to implementers.

The implementation should make a best effort to free an invalidated communicator locally and return MPI_SUCCESS. Otherwise, it must return MPI_ERR_INVALIDATED.

Note for the Forum. The text of Page 208 lines 39-48 must be amended to provide the following advice to users.

Because MPI_COMM_FREE resets the MPI_Errhandler of a communicator to MPI_ERRORS_ARE_FATAL, fault tolerant applications should complete all pending communications before calling MPI_COMM_FREE.

MPI_COMM_SHRINK(comm, newcomm)

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

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

This function must not return an error due to process failures (error classes MPI_ERR_PROC_FAILED and MPI_ERR_INVALIDATED). Upon successful completion, an agreement is made among living processes to determine the group of failed processes. This group includes at least all processes whose failure has been notified to the user. The call is semantically equivalent to MPI_COMM_SPLIT, where living processes participate with the same color, and a key equal to their rank in *comm* and failed processes implicitly contribute MPI_UNDEFINED.

Advice to users. This call does not guarantee that all processes in *newcomm* are alive. Any new failure will be detected in subsequent MPI calls. (End of advice to users.)

MPI_COMM_FAILURE_ACK(comm)

IN	comm	**communicator (handle)**

This local function gives the users a way to acknowledge all locally notified failures on *comm*. After the call, unmatched MPI_ANY_SOURCE receptions that would have returned an error code due to process failure (see Section 1.2.1) proceed without further reporting of errors due to acknowledged failures.

Advice to users. Calling MPI_COMM_FAILURE_ACK on a communicator with failed processes does not allow that communicator to be used successfully for collective operations. Collective communication on a communicator with acknowledged failures will continue to return an error of class MPI_ERR_PROC_FAILED as defined in Section 1.2.1. To reliably use collective operations on a communicator with failed processes, the communicator should first be invalidated using MPI_COMM_INVALIDATE and then a new communicator should be created using MPI_COMM_SHRINK. (End advice to users.)

MPI_COMM_FAILURE_GET_ACKED(comm, failedgroup)

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

This local function returns the group *failedgroup* of processes, from the communicator *comm*, which have been locally acknowledged as failed by preceding calls to MPI_COMM_FAILURE_ACK.

MPI_COMM_AGREEMENT(comm, flag)

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

This function performs a collective operation among all living processes in *comm*. On completion, all living processes must agree to set the value of *flag* to the result of a logical *'AND'* operation over the contributed values. This function must not return an error due to process failure (error classes MPI_ERR_PROC_FAILED and MPI_ERR_INVALIDATED), and failed processes do not contribute to the operation.

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

Advice to users. MPI_COMM_AGREEMENT maintains its collective meaning even if the *comm* is invalidated. (End of advice to users.)

MPI_ICOMM_AGREEMENT(comm, flag, req)

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

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

1.3.2 One-Sided Functions

MPI_WIN_INVALIDATE (win)

IN	win	**window (handle)**

This function notifies all ranks within the window *win* that this window is now considered invalid. An invalid window preempts any non-local MPI calls on *win*. Once a window has been invalidated, all subsequent non-local calls on that window are considered local and must fail with an error of class MPI_ERR_INVALIDATED.

MPI_WIN_GET_FAILED(win, failedgroup)

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

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

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

Advice to users. It is possible that only the calling process has detected the reported failure. If global knowledge is necessary, processes detecting failures should use the call MPI_WIN_INVALIDATE. (End advice to users.)

1.3.3 I/O Functions

MPI_FILE_INVALIDATE (fh)

IN	fh	**file (handle)**

This function eventually notifies all ranks within file *fh* that this file is now considered invalid. An invalid file preempts any non-local completion calls on *file* (see Section 1.2.4). Once a file has been invalidated, all subsequent non-local calls on the file must fail with an error of class MPI_ERR_INVALIDATED.

1.4 Error Codes and Classes

MPI_ERR_PROC_FAILED	A process in the operation has failed (a fail-stop failure).
MPI_ERR_INVALIDATED	The communication object used in the operation was invalidated.