Chapter 17

Process Fault Tolerance

17.1 Introduction

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

The expected behavior of MPI in the case of a process failure is defined by the following statements: any MPI operation that involves a failed process must not block indefinitely, but either succeed or raise an MPI exception (see Section 17.2); an MPI operation that does not involve the failed process will complete normally, unless interrupted by the user through provided functionality. Asynchronous failure propagation is not required. If an application needs global knowledge of failures, it can use the interfaces defined in Section 17.3 to explicitly propagate locally detected failures.

This chapter does not define process failure semantics for the operations specified in Chapters [10,][11][and 13], therefore they remain undefined by the MPI standard. ticket327.

An implementation that does not tolerate process failures must provide the interfaces ticket325.
and semantics defined in this chapter, but must never raise an exception of class ticket326.
MPI_ERR_PROC_FAILED or MPI_ERR_PENDING related to process failure (as defined below). ticket0.

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.*)

17.2 Failure Notification

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

1. the operation is collective and the process appears in one of the groups of the associated communication object;

2. the process is a specified or matched destination or source in a point-to-point communication;
3. the operation is an MPI_ANY_SOURCE receive operation and the failed process belongs to the source group.

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

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

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

17.2.1 Startup and Finalize

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

ticket0. MPI_FINALIZE will complete [succesfully]successfully even in the presence of process failures.

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

17.2.2 Point-to-Point and Collective Communication

ticket0. [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. Future point-to-point communication with the same process on this communicator must also return MPI_ERR_PROC_FAILED.

The completion of a nonblocking receive from MPI_ANY_SOURCE can return one of the following three error codes due to process failure. MPI_SUCCESS is returned if the receive was able to complete despite the failure. MPI_ERR_PROC_FAILED indicates that the request has been matched with the send, but cannot complete [succesfully]successfully due to the failure at the sender. MPI_ERR_PENDING indicates that while a process has failed, the request is still pending and can be continued. To acknowledge a failure and discover which processes failed, the user should call MPI_COMM_FAILURE_ACK.

ticket0.

Advice to implementors.

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 as a named receive. If the operation has not yet matched and was initiated 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.

(*End of advice to implementors.*)

] ticket0.

When the failure of a process involved in a communication operation is discovered by the MPI implementation before the successful completion of the operation, the communication completion function must raise one of the following error classes:

- MPI_ERR_PENDING indicates that the communication is a non-blocking operation and neither the operation nor the request identifying the operation are completed. Two circumstances can raise this exception: another communication raised an exception (as defined in Section 3.7.5); or the communication is a receive operation from MPI_ANY_SOURCE and no matching send has been posted.
- In all other cases, the operation must raise an exception of class MPI_ERR_PROC_FAILED which indicates that the failure prevents the operation from following its failure-free specification. If there is a request identifying the communication operation, it is completed.

Advice to users.

To acknowledge a failure and discover which processes failed, the user should call MPI_COMM_FAILURE_ACK (as defined in Section 17.3.1).

(*End of advice to users.*)

When a communication operation raises an exception related to process failure, any output buffers are *undefined*.

When a collective operation cannot be completed because of the failure of an involved process, the collective operation [eventually] returns an error of class ticket0. MPI_ERR_PROC_FAILED. [The content of the output buffers is *undefined*.] ticket0.

Advice to users.

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

(*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 have each process either revoke (see Section 17.3.1) the parent communicator if the operation fails, or call an MPI_BARRIER on the parent communicator and then revoke the new communicator if the MPI_BARRIER fails.

(*End of advice to users.*)

ticket327.

17.2.3 Dynamic Process Management

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

1. If the MPI implementation returns an error related to process failure to the root process of MPI_COMM_CONNECT or MPI_COMM_ACCEPT, at least the root processes of both intracommunicators must return the same error of class MPI_ERR_PROC_FAILED (unless required to return MPI_ERR_REVOKED as defined by 17.3.1).
2. If the MPI implementation returns an error related to process failure to the root process of MPI_COMM_SPAWN, no spawned processes should be able to communicate on the created intercommunicator.

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

ticket325.

17.2.4 One-Sided Communication

As with all nonblocking operations, one-sided communication operations should delay all failure notification until their synchronization operations which may return MPI_ERR_PROC_FAILED (see Section 17.2). If the implementation returns an error related to process failure from the synchronization function, the epoch behavior is unchanged from the definitions in Section 11.4. As with collective operations over MPI communicators, it is possible that some processes have detected a failure and returned MPI_ERR_PROC_FAILED, while others returned MPI_SUCCESS.

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

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

2. If the target rank has failed, MPI_WIN_LOCK and MPI_WIN_UNLOCK operations 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.*)

ticket326.

17.2.5 I/O

I/O error classes and their consequences are defined in [s]Section 13.7. The following section defines the behavior of I/O operations when MPI process failures prevent their successful completion. ticket0.

Since collective I/O operations may not synchronize with other processes, process failures may not be reported during a collective I/O operation. If a process failure prevents a file operation from completing, an MPI exception of class MPI_ERR_PROC_FAILED is raised.

Once an MPI implementation has returned an error of class MPI_ERR_PROC_FAILED, the state of the file pointer is *undefined*.

> *Advice to users.*
>
> Users are encouraged to use MPI_COMM_AGREE on a communicator containing the same group as the file handle, to deduce the completion status of collective operations on file handles and maintain a consistent view of file pointers.
>
> (*End of advice to users.*)

17.3 Failure Mitigation Functions

17.3.1 Communicator Functions

MPI provides no guarantee of global knowledge of a process failure. Only processes involved in a communication operation with the failed process are guaranteed to eventually detect its failure (see Section 17.2). If global knowledge is required, MPI provides a function to revoke a communicator at all members.

MPI_COMM_REVOKE(comm)

IN	comm	communicator (handle)

```
int MPI_Comm_revoke(MPI_Comm comm)
```

```
MPI_COMM_REVOKE(COMM, IERROR)
    INTEGER COMM, IERROR
```

This function notifies all processes in the groups (local and remote) associated with the communicator comm that this communicator is now considered revoked. This function is not collective and therefore does not have a matching call on remote processes. It is erroneous to call MPI_COMM_REVOKE on a communicator for which no operation raised an MPI exception related to process failure. All alive processes belonging to comm will be notified of the revocation despite failures. Revocation of a communicator completes any non-local MPI operations on comm with error and causes any new operations to complete with error, with the exception of MPI_COMM_SHRINK and MPI_COMM_AGREE (and its nonblocking equivalent). A communicator becomes revoked as soon as: ticket0.

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

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

Advice to users. High quality implementations are encouraged to do their best to free resources locally when the user calls free operations on revoked communication objects, or communication objects containing failed processes. (*End of advice to users.*)

MPI_COMM_SHRINK(comm, newcomm)

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

```
int MPI_Comm_shrink(MPI_Comm comm, MPI_Comm* newcomm)
```

```
MPI_COMM_SHRINK(COMM, NEWCOMM, IERROR)
    INTEGER COMM, NEWCOMM, IERROR
```

This collective operation creates a new intra or inter communicator newcomm from the revoked intra or inter communicator comm respectively by excluding its failed processes as detailed below. It is erroneous MPI code to call MPI_COMM_SHRINK on a communicator which has not been revoked (as defined above) and will 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_REVOKED). All processes that succeeded agreed on the content of the group of processes that failed. This group includes at least every process failure that has raised an MPI exception of class MPI_ERR_PROC_FAILED or MPI_ERR_PENDING. The call is semantically equivalent to an MPI_COMM_SPLIT operation that would succeed despite failures, and where living processes participate with the same color, and a key equal to their rank in comm and failed processes implicitly contribute MPI_UNDEFINED.

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

MPI_COMM_FAILURE_ACK(comm)

IN	comm	communicator (handle)

```
int MPI_Comm_failure_ack(MPI_Comm comm)
```

```
MPI_COMM_FAILURE_ACK(COMM, IERROR)
    INTEGER COMM, IERROR
```

This local operation gives the users a way to *acknowledge* all locally notified failures on comm. After the call, unmatched MPI_ANY_SOURCE receptions that would have returned an error code due to process failure (see Section 17.2.2) proceed without further reporting of errors due to those acknowledged failures.

> *Advice to users.* Calling MPI_COMM_FAILURE_ACK on a communicator with failed processes does not allow that communicator to be used successfully for collective operations. Collective communication on a communicator with acknowledged failures will continue to return an error of class MPI_ERR_PROC_FAILED as defined in Section 17.2.2. To reliably use collective operations on a communicator with failed processes, the communicator should first be revoked using MPI_COMM_REVOKE and then a new communicator should be created using MPI_COMM_SHRINK. (*End of advice to users.*)

MPI_COMM_FAILURE_GET_ACKED(comm, failedgrp)

IN	comm	communicator (handle)
OUT	failedgrp	group of failed processes (handle)

```
int MPI_Comm_failure_get_acked(MPI_Comm comm, MPI_Group* failedgrp)
```

```
MPI_COMM_FAILURE_GET_ACKED(COMM, FAILEDGRP, IERROR)
    INTEGER COMM, FAILEDGRP, IERROR
```

This local operation returns the group failedgrp of processes, from the communicator comm, which have been locally acknowledged as failed by preceding calls to MPI_COMM_FAILURE_ACK. [The new group]failedgrp can be empty, that is, equal to MPI_GROUP_EMPTY. ticket0.

MPI_COMM_AGREE(comm, flag)

IN	comm	communicator (handle)
INOUT	flag	boolean flag

```
int MPI_Comm_agree(MPI_Comm comm, int * flag)
```

```
MPI_COMM_AGREE(COMM, FLAG, IERROR)
    LOGICAL FLAG
    INTEGER COMM, IERROR
```

This function performs a collective operation on the group of living processes in comm. On completion, all living processes must agree to set the output value of flag to the result of ticket0. a logical *'AND'* operation over the in[t]put values of flag. This function must not return an error due to process failure (error classes MPI_ERR_PROC_FAILED and MPI_ERR_REVOKED), and processes that failed before entering the call do not contribute to the operation.

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

> *Advice to users.* MPI_COMM_AGREE maintains its collective behavior even if the comm is revoked. (*End of advice to users.*)

MPI_COMM_IAGREE(comm, flag, req)

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

```
int MPI_Comm_iagree(MPI_Comm comm, int* flag, MPI_Request* req)
```

```
MPI_COMM_IAGREE(COMM, FLAG, REQ, IERROR)
    LOGICAL FLAG
    INTEGER COMM, REQ, IERROR
```

This function has the same semantics as MPI_COMM_AGREE except that it is non-ticket325. blocking.

17.3.2 One-Sided Functions

MPI_WIN_REVOKE(win)

IN	win	window (handle)

```
int MPI_Win_revoke(MPI_Win win)
```

```
MPI_WIN_REVOKE(WIN, IERROR)
    INTEGER WIN, IERROR
```

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

MPI_WIN_GET_FAILED(win, failedgrp)

IN	win	window (handle)
OUT	failedgrp	group of failed processes (handle)

```
int MPI_Win_get_failed(MPI_Win win, MPI_Group* failedgrp)

MPI_WIN_GET_FAILED(WIN, FAILEDGRP, IERROR)
    INTEGER COMM, FAILEDGRP, IERROR
```

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

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

ticket326.

17.3.3 I/O Functions

MPI_FILE_REVOKE(fh)

IN	fh	file (handle)

```
int MPI_File_revoke(MPI_File fh)

MPI_FILE_REVOKE(FH, IERROR)
    INTEGER FH, IERROR
```

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

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

17.4 Error Codes and Classes

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

17.5 Examples

17.5.1 Master/Worker

The example below presents a master code that handles failures by ignoring failed processes and resubmitting requests. It demonstrates the different failure cases that may occur

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

Table 17.1: Additional process fault tolerance error classes

when posting receptions from MPI_ANY_SOURCE as discussed in the advice to users in Section 17.2.2.

Example 17.1 Fault-Tolerant Master Example

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

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

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

    /* Progress engine: Get answers, send new requests,
       and handle process failures */
    while( (active_workers > 0) && work_available ) {
        rc = MPI_Wait( &req, &status );

        if( (MPI_ERR_PROC_FAILED == rc) || (MPI_ERR_PENDING == rc) ) {
            MPI_Comm_failure_ack(comm);
            MPI_Comm_failure_get_acked(comm, &g);
            MPI_Group_size(g, &gsize);

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

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

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

            continue;
        }

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

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

17.5.2 Iterative Refinement

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

Example 17.2 Fault-tolerant iterative refinement with shrink and agreement

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

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

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

        /* About to leave: let's be sure that everybody
           received the same information */
        allsucceeded = (rc == MPI_SUCCESS);
        MPI_Comm_agree(comm, &allsucceeded);
        if( !allsucceeded ) {
            /* We plan to join the shrink, thus the communicator
               should be marked as revoked */
            MPI_Comm_revoke(comm);
            MPI_Comm_shrink(comm, &comm2);
            MPI_Comm_free(comm); /* Release the revoked communicator */
            comm = comm2;
            gnorm = epsilon + 1.0; /* Force one more iteration */
        }
    }
}
```