# *D R A F T*
# Document for a Standard Message-Passing Interface

Message Passing Interface Forum

January 22, 2014
This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

# Chapter 17

# Process Fault Tolerance

## 17.1 Introduction

In distributed systems with numerous or complex components, a serious risk is that a component fault manifests as a process failure that disrupts the normal execution of a long running application. A process failure is a common outcome for many hardware, network, or software faults that cause a process to crash; It can be more formally defined as a fail-stop failure: the failed process becomes permanently unresponsive to communications. This chapter introduces MPI features that support the development of applications, libraries, and programming languages that can tolerate process failures. The primary goal is to specify error classes and interfaces that permit users to continue simple MPI communication operations after failures have impacted the execution, and rebuild MPI objects (communicators, files, etc.) as needed to restore the full capability of MPI to carry elaborate communication operations (like collective communications). This specification does not include mechanisms to restore the lost data from failed processes; the literature is rich with diverse fault tolerance techniques that the users may employ at their discretion, including checkpoint-restart, algorithmic dataset recovery, and continuation ignoring failed processes. All these fault tolerance approaches benefit from, and often require, the definitions and interfaces specified in this chapter in order to resume communicating after a failure.

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 a failed process will complete normally, unless interrupted by the user through provided functionality. Exceptions indicate only the local impact of the failure on an operation, and make no guarantee that other processes have also been notified of the same failure. Asynchronous failure propagation is not guaranteed or required, and users must exercise caution when reasoning on the set of ranks where a failure has been detected and raised an exception. If an application needs global knowledge of failures, it can use the interfaces defined in Section 17.3 to explicitly propagate the notification of locally detected failures.

The typical usage pattern on some reliable machines may not require fault tolerance. An MPI implementation that does not tolerate process failures must never raise an exception of class MPI_ERR_PROC_FAILED, MPI_ERR_REVOKED, or MPI_ERR_PROC_FAILED_PENDING to report a process failure. Fault-tolerant applications using the interfaces defined in this chapter must compile, link, and run successfully in failure-free executions.

*Advice to users.* Many of the operations and semantics described in this chapter are applicable only 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.

An operation involving a failed process must always complete in a finite amount of time (possibly by raising a process failure exception). If an operation does not involve a failed process (such as a point-to-point message between two nonfailed processes), it must not raise a process failure exception.

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

When a communication operation raises an exception related to process failure, it may not satisfy its specification. In particular, a synchronizing operation may not have synchronized, and the content of the output buffers, targeted memory, or output parameters is *undefined*. Operations excepting this rule are explicitly stated in the remainder of this chapter.

Non-blocking operations must not raise an exception 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.*)

MPI_FINALIZE will complete successfully even in the presence of process failures.

*Advice to users.* MPI raises exceptions only before MPI_FINALIZE is invoked and thereby 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. In Example 8.10 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, possibly leading to this code never being executed. (*End of advice to users.*)

### 17.2.2 Point-to-Point and Collective Communication

An MPI implementation raises exceptions of the following error classes in order to notify users that a point-to-point communication operation could not complete successfully because of the failure of involved processes:

- MPI_ERR_PROC_FAILED_PENDING indicates, for a non-blocking communication, that the communication is a receive operation from MPI_ANY_SOURCE and no matching send has been posted, yet a potential sender process has failed. Neither the operation nor the request identifying the operation are completed.
- In all other cases, the operation raises an exception of class MPI_ERR_PROC_FAILED to indicate that the failure prevents the operation from following its failure-free specification. If there is a request identifying the point-to-point communication, it is completed. Future point-to-point communication with the same process on this communicator must also raise MPI_ERR_PROC_FAILED.

*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 collective operation cannot be completed because of the failure of an involved process, the collective operation raises an exception of class MPI_ERR_PROC_FAILED.

*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 raising an exception.

Note, however, for some operations' semantics, when a process fails before entering the operation, it forces raising an exception at all ranks. As an example, if an operation on an intracommunicator has raised an exception, the process receiving that exception can then assume that in a subsequent MPI_BARRIER on this communicator, all ranks will raise an exception MPI_ERR_PROC_FAILED because the participating process is known to have failed before entering the barrier.

(*End of advice to users.*)

*Advice to users.*

Note that communicator creation functions (e.g., MPI_COMM_DUP or MPI_COMM_SPLIT) are collective operations. As such, if a failure happened during the call, an exception might be raised at some processes while others succeed and obtain a new communicator. Although it is valid to communicate between processes that succeeded in creating the new communicator, the user is responsible for ensuring a consistent view of the communicator creation, if needed. A conservative solution is check the global outcome of the communicator creation function with MPI_COMM_AGREE (defined in Section 17.3.1), as illustrated in Example 17.1. (*End of advice to users.*)

After a process failure, MPI_COMM_FREE (as with all other collective operations) may not complete successfully at all ranks. For any rank that receives the return code MPI_SUCCESS, the behavior is defined as in Section 6.4.3. If a rank raises a process failure exception (MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED), the implementation makes no guarantee about the success or failure of the MPI_COMM_FREE operation remotely; however, it still attempts to clean up any local data used by the communicator object. This will be signified by returning MPI_COMM_NULL only when the object has successfully been freed locally.

### 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 raises an exception 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 raise the same exception of class MPI_ERR_PROC_FAILED (unless required to raise MPI_ERR_REVOKED as defined in Section 17.3.1). The same is true if the implementation raises an exception at any process in MPI_COMM_JOIN.

2. If the MPI implementation raises an exception related to process failure to the root process of MPI_COMM_SPAWN or MPI_COMM_SPAWN_MULTIPLE, no spawned processes should be able to communicate on the created intercommunicator.

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

### 17.2.4 One-Sided Communication

One-sided communication operations must provide failure notification in their synchronization operations that may raise an exception due to process failure (see Section 17.2). If the implementation does not raise an exception related to process failure in the synchronization function, the epoch behavior is unchanged from the definitions in Section 11.5. As with collective operations over MPI communicators, some processes may have detected a failure and raised MPI_ERR_PROC_FAILED, while others returned MPI_SUCCESS. Once the implementation raises an exception related to process failure on a specific window in a synchronization function, all subsequent operations on that window must also raise an exception related to process failure.

Unless specified below, the state of memory targeted by any process in an epoch in which operations raised an exception related to process failure is undefined, with the exception of memory targeted by remote read operations (and operations which are semantically equivalent to read operations, such as an MPI_GET_ACCUMULATE with MPI_NO_OP as the operation). All other window locations are valid.

If an exception is raised from an active target synchronization operation 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.

MPI_WIN_LOCK and MPI_WIN_UNLOCK may raise MPI_ERR_PROC_FAILED when any process in the window has failed. An implementation cannot block indefinitely in a correct program waiting for a lock to be acquired; If the owner of the lock has failed, some other process trying to acquire the lock either succeeds or raises an exception of class MPI_ERR_PROC_FAILED. If the target rank has failed, MPI_WIN_LOCK and MPI_WIN_UNLOCK operations must raise an exception of class MPI_ERR_PROC_FAILED. The lock cannot be acquired again at any target in the window, and all subsequent operations on the lock must raise MPI_ERR_PROC_FAILED.

> *Advice to implementors.* If a nontarget rank in the window fails, a high-quality implementation may be able to mask such a fault inside the locking algorithm and continue to allow the remaining ranks to acquire the lock without raising errors. (*End of advice to implementors.*)

It is possible that request-based RMA operations complete successfully (via operations such as MPI_TEST or MPI_WAIT) while the enclosing epoch completes by raising an exception due to a process failure. In this scenario, the local buffer is valid, but the remote targeted memory is undefined.

After a process failure, MPI_WIN_FREE (as with all other collective operations) may not complete successfully at all ranks. For any rank that receives the return code MPI_SUCCESS, the behavior is defined as in Section 11.2.5. If a rank raises a process failure exception (MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED), the implementation makes no guarantee about the success or failure of the MPI_WIN_FREE operation remotely; however, it still attempts to clean up any local data used by the window object. This will be signified by returning MPI_WIN_NULL only when the object has successfully been freed locally.

### 17.2.5 I/O

This section defines the behavior of I/O operations when MPI process failures prevent their successful completion. I/O backend failure error classes and their consequences are defined in Section 13.7.

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 raised an exception of class MPI_ERR_PROC_FAILED, the state of the file pointer involved in the operation that raised the exception is *undefined*.

*Advice to users.* Since collective I/O operations may not synchronize with other processes, process failures may not be reported during a collective I/O operation. Users are encouraged to use MPI_COMM_AGREE on a communicator containing the same group as the file handle when they need to deduce the completion status of collective operations on file handles and maintain a consistent view of file pointers. The file pointer can be reset by using MPI_FILE_SEEK with the MPI_SEEK_SET update mode. (*End of advice to users.*)

After a process failure, MPI_FILE_CLOSE (as with all other collective operations) may not complete successfully at all ranks. For any rank that receives the return code MPI_SUCCESS, the behavior is defined as in Section 13.2.2. If a rank raises a process failure exception (MPI_ERR_PROC_FAILED or MPI_ERR_REVOKED), the implementation makes no guarantee about the success or failure of the MPI_FILE_CLOSE operation remotely; however, it still attempts to clean up any local data used by the file handle. This will be signified by returning MPI_FILE_NULL only when the object has successfully been freed locally.

## 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. All alive processes belonging to comm will be notified of the revocation despite failures. The revocation of a communicator completes any non-local MPI operations on comm by raising an exception of class MPI_ERR_REVOKED, with the exception of MPI_COMM_SHRINK and MPI_COMM_AGREE (and its nonblocking equivalent). A communicator becomes revoked as soon as either of the following occur:

1. MPI_COMM_REVOKE is locally called on it;
2. Any MPI operation raised an exception 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 by raising an exception 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 intercommunicator newcomm from the intra- or intercommunicator comm, respectively, by excluding its failed processes, as detailed below. It is valid MPI code to call MPI_COMM_SHRINK on a communicator that has been revoked (as defined above).

This function must not raise an exception due to process failures (error classes MPI_ERR_PROC_FAILED and MPI_ERR_REVOKED). All processes agree 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_PROC_FAILED_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 raised an exception MPI_ERR_PROC_FAILED_PENDING due to process failure (see Section 17.2.2) proceed without further raising exceptions 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 raise an exception of class MPI_ERR_PROC_FAILED as defined in Section 17.2.2. In order to resume using collective operations when a communicator contains failed processes, a new communicator should be created by 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, that have been locally acknowledged as failed by preceding calls to MPI_COMM_FAILURE_ACK. The *failedgrp* can be empty, that is, equal to MPI_GROUP_EMPTY.

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 agree to set the output value of flag to the result of a logical *'AND'* operation over the contributed input values of flag. 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).

If the flag result ignores the contribution of a failed process, and this failure has not been acknowledged by a prior call to MPI_COMM_FAILURE_ACK, MPI_COMM_AGREE raises an exception of class MPI_ERR_PROC_FAILED; when such an exception is raised at any rank, it is raised at all ranks consistently. The value of flag remains correct when this exception is raised.

When MPI_COMM_AGREE completes, the failure of any process that failed before it entered the call to MPI_COMM_AGREE can be acknowledged by a following call to MPI_COMM_FAILURE_ACK.

This function never raises an exception of class MPI_ERR_REVOKED.

*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-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. This function is not collective and therefore does not have a matching call on remote processes. All alive processes belonging to win will be notified of the revocation despite failures. The revocation of a window completes any non-local MPI operations on win by raising an exception of class MPI_ERR_REVOKED. Once a window has been revoked, all subsequent non-local operations on that window are considered local and must raise an exception 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 that 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 update only the group of locally known failed processes when it enters a synchronization function and must raise a process failure exception. (*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.*)

### 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 processes within the file handle fh that this file handle is now considered revoked. This function is not collective and therefore does not have a matching call on remote processes. All alive processes belonging to the file handle fh will be notified of the revocation despite failures. The revocation of a file handle completes any non-local MPI operations on win by raising an exception of class MPI_ERR_REVOKED. Once a file handle has been revoked, all subsequent non-local operations on that file handle are considered local and must raise an 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:

| | |
|---|---|
| MPI_ERR_PROC_FAILED | The operation could not complete because of a process failure (a fail-stop failure). |
| MPI_ERR_PROC_FAILED_PENDING | The operation was interupted by a process failure (a fail-stop failure). The request is still pending and the operation may be completed later. |
| MPI_ERR_REVOKED | The communication object used in the operation has been revoked. |

Table 17.1: Additional process fault tolerance error classes

## 17.5 Examples

### 17.5.1 Safe Communicator Creation

The example below illustrates how a new communicator can be safely created despite disruption by process failures. A child communicator is created with MPI_COMM_SPLIT, then the global success of the operation is verified with MPI_COMM_AGREE. If any process failed to create the child communicator, all processes are notified by the value of the boolean flag agreed on. Processes that had successfully created the child communicator destroy it, as it cannot be used consistently.

**Example 17.1** Fault Tolerant Communicator Split Example

```
int Comm_split_consistent(MPI_Comm parent, int color, int key, MPI_Comm* child)
{
    rc = MPI_Comm_split(parent, color, key, child);
    split_ok = (MPI_SUCCESS == rc);
    MPI_Comm_agree(parent, &split_ok);
    if(split_ok) {
        /* All surviving processes have created the "child" comm
         * It may contain supplementary failures and the first
         * operation on it may raise an exception, but it is a
         * workable object that will yield well specified outcomes */
        return MPI_SUCCESS;
    }
    else {
        /* At least one process did not create the child comm properly
         * if the local rank did succeed in creating it, it disposes
         * of it, as it is a broken, inconsistent object */
        if(MPI_SUCCESS == rc) {
            MPI_Comm_free(child);
        }
        return MPI_ERR_PROC_FAILED;
    }
}
```

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

**Example 17.2** 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_FAILURE_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 ... */
}
```

### 17.5.3 Iterative Refinement

The example below demonstrates a method of fault tolerance for detecting and handling 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 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 17.3** 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) ) {

        /* This rank detected a failure, but other ranks may have
         * proceeded into the next MPI_Allreduce. Since this rank
         * will not match that following MPI_Allreduce, these other
         * ranks would be at risk of deadlocking. This process thus
         * calls MPI_Comm_revoke to interrupt other ranks and notify
         * them that it has detected a failure and is leaving the
         * failure free execution path to go into recovery. */
        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);
        rc = MPI_Comm_agree(comm, &allsucceeded);
        if( rc == MPI_ERR_PROC_FAILED || !allsucceeded ) {
            MPI_Comm_shrink(comm, &comm2);
            MPI_Comm_free(comm); /* Release the revoked communicator */
            comm = comm2; gnorm = epsilon + 1.0; /* Force one more iteration */
        }
    }
}
```

# Index