MPI Error Reporting Rules

By Erez Haba

Abstract

The focus of this document is to define MPI error reporting rules in a way that better serves library and frameworks that are using MPI, while maintaining compatibility with existing applications error handling code. These rules are essential for fault-tolerant support in MPI and are building blocks for this effort. The suggestions presented hereby sprung from the discussion in fault-tolerance working group about the need for correct error reporting when handling reporting and handling faults.

The documents then goes further and based on the concepts introduced in this paper, suggest a mechanism to handle consistent error reporting across all ranks for collective operations. This suggestion takes somewhat a different approach and put the burden of checking for consistency on the application programmer rather than the MPI library collective implementation.

Description

The MPI 2.1 standard provides just a general concept on how and when an MPI implementation should return an error once detected. As a result in many MPI implementations the error is reported back to the application as soon as it is detected, and in many occasions result in application termination. Errors are reported by MPI implementations even if they are not related to the MPI call site and might be not relevant to that specific call. This makes error handling in the context of the call site more complicated and leads to the default behavior of bailing out on error.

The main reason for this behavior is the MPI progress engine. The progress engine processes events from different sources in the system and for different active request, usually without associating the event with the “current” or “right” request. As an error occurs it simplify returns “an error happen” which then get returned to the immediate call site, rather than being associated with the actual request being affected.

What I would like to introduce here is the concept of error separation and associatively to the actual logical request. These concepts are in line with the existing rules and are compatible with existing applications. The definition is rather simple, identifying when in an event of an error it should be returned (and an error handler should be called) and when the error return should be postponed.

Error Reporting Rules

Here are the few directives and rules I came up with to define when an error should be returned. I'll start by introducing few concepts,

1. *Virtual Connection*
   Virtual Connection represents the immediate link between any two ranks. I assume that an error can be detected on a virtual connection related to either network or process failure (or other reasons for failure). A virtual connection moves to an "error state" once an error has been detected, both sides of the link do not have to agree on that state.

2. *Communicator*
   This is not a new concept; it is the MPI communicator extended with a state. A communicator moves to an "error state" once any of its virtual connections has moved to an error state.

Rules

Only few rules to follow to meet this error return model

1. A virtual connection moves to an error state, once unable to communicate with its peer.
2. A virtual connection in an error state cause pending or future requests associated with it and only with it to move to an error state. That is immediate calls, pending requests or future calls.
3. A virtual connection in an error state moves all communicators associated with it to an error state
4. A communicator in an error state affects only pending or future receives with MPI_ANY_SOURCE
5. Errors are returned per call site, and associated with the call context.

Examples

All examples are running on rank 0, unless noted otherwise.

```
//
// Example e1:
// Comm error detected with rank 4 while receive from that rank is pending.
//

MPI_Irecv(source=4, &request4);   // returns success

//
// Error in the communication with rank4 detected. The communicator moves to
// an error state, but the send call returns success.
//
MPI_Send(dest=3);                 // returns success

//
// The error is associated with request4 and thus returned in the MPI_Wait call
//
MPI_Wait(request4);               // returns fail
```

```
//
// Example e2:
// Comm error detected with rank 4 while a receive from any rank is pending.
//

MPI_Irecv(source=any, &request); // returns success

//
// Error in the communication with rank4 detected. The communicator moves to
// an error state, but the send call returns success.
//
MPI_Send(dest=3);                 // returns success

//
// The communicator set to the error state, any receives with MPI_ANY_SOURCE
// are set to error state and thus return an error in their wait call
//
MPI_Wait(request);                // returns fail
```

```
//
// Example e3:
// Comm error detected with rank 4 before calling receive/send.
//

//
// Error in the communication with rank4 detected, the communicator moves to
// an error state, but the send call to rank 3 returns success
//
MPI_Send(dest=3);                  // returns success

//
// The communicator set to the error state, any receives with MPI_ANY_SOURCE
// are set to error state and thus return an error.
//
MPI_Recv(source=any);              // returns fail

//
// VC 4 is set to the error state, any usage of that VC will return an error.
//
MPI_Recv(source=4);                // returns fail
MPI_Send(dest=4);                  // returns fail
```

```
//
// Example e4:
// Comm error detected with rank 4 while a bcast(root=0) call is in progress.
//

MPI_Irecv(source=any, &request); // returns success

//
// Error in the communication with rank4 detected. The communicator moves to
// an error state, however bcast is not using rank 4 directly in its
// communication and thus returns success.
//
MPI_Bcast(root=0);                 // returns success

//
// The communicator set to the error state, any receives with MPI_ANY_SOURCE
// are set to error state and thus return an error in their wait call
//
MPI_Wait(request);                 // returns fail
```

```
//
// Example e5:
// Comm error detected with rank 4 while a bcast(root=3) call is in progress.
//

MPI_Irecv(source=any, &request);  // returns success

//
// Error in the communication with rank4 detected. The communicator moves to
// an error state, bcast is using rank 4 directly in its communication and
// thus returns fail.
//
MPI_Bcast(root=3);                // returns fail

//
// The communicator set to the error state, any receives with MPI_ANY_SOURCE
// are set to error state and thus return an error in their wait call
//
MPI_Wait(request);                // returns fail
```

TODO:

File API's

RMA API's

Semantic and example with errors in some collectives like,

MPI_Comm_split

MPI_Inercomm_merge

Usage Scenarios

Suggestion Impact

Standard

Users

Implementers

Collective Validation

As can be seen in examples 4 & 5 above the error results from a collective call might not be consistent across ranks and across runs. This creates a problem as the code following the collective operation takes action base on the success/fail result of the operation. This might take different ranks on different code path which might end up with an inconsistent state across the communicator and hence a deadlock/hang.

The proposal in this section takes the approach of putting this consistency check on the programmer rather than extending (and affecting) the collectives implementation to return consistent error result across all ranks.

The suggested solution here is a new collective API that checks the communicator validity across all participating ranks. If any of the communicators is an error state as described above an error is returned to all callers; if an error is detected during the validation collective, an error is returned to all ranks in that communicator. There are few challenges in implementing this interface in an effective way, but the basic two-phase commit implementation seems to suffice.

```
//
// Validate communicator consistency across all ranks in the communicator
//
int MPI_Comm_validate(MPI_COMM comm);
```

This collective API returns a communicator validity indicator only for completed operations. That is if invoked after a blocking call, it will indicate that the previous operation was successful or not; however if invoked after an incomplete asynchronous operation, a success value does not give any indication as to the success of the incomplete asynchronous operations.

```
//
// Example v1:
// Validate the successful completion of a bcast collective
//

//
// Error in the communication with rank4 detected. The communicator moves to
// an error state, bcast is using rank 4 directly in its communication and
// thus returns fail. However other ranks in this collective operations return
// success as they did not communicate with rank 4.
//
MPI_Bcast(root=3, comm);          // returns fail (on some ranks)

//
// Validate the previous collective on all ranks. Since rank0 detected an error
// with rank 3, and since rank 3 might not respond; all ranks will detect this
error.
//
MPI_Comm_validate(comm);          // returns fail (on ALL ranks)
```

```
//
// Example v2:
// Validate the successful completion of a send/receive
//

//
// Error in the communication with rank 1 detected. The rank 0 communicator
moves
// to an error state, as well as rank 2; however other ranks will not
//
MPI_Isend(dst= rank+1 % size, comm, &request[0]);
MPI_Irecv(src= rank-1 % size, comm, requests[1]);
MPI_Waitall(2, &requests);              // returns fail (for ranks 0, 1, 2)

//
// Validate the previous completed communication by all ranks in this
communicator.
// Since rank0 detected an error; all ranks will detect this error.
//
MPI_Comm_validate(comm);          // returns fail (on ALL ranks)
```