Proposed API in support Fault Tolerance in MPI

MPI-3 Fault Tolerance Working Group

October 16, 2009

Contents

0.1 Introduction

This chapter introduces MPI features that support the development of fault tolerant applications. Fault tolerant actions described herein are constrained to that which is directly under the control of MPI. Thus, for example, recovery of lost data is *not* the responsibility of MPI. We refer the reader to [3, 2] for further information on writing fault tolerant applications using the features described in this chapter.

0.1.1 Assumptions

Support for fault tolerance is incorporated into MPI under the following assumptions:

- Backward compatibility is required.
- Errors are associated with specific call sites.
- An application may choose to be notified when an error occurs anywhere in the system.
- An application may ignore failures that do not impact its MPI requests.
- An MPI process may ignore failures that do not impact its MPI requests[1].

[1] The user should understand the potential for unsatisfied requests caused by dependency chains of communication among processes.

- An application that does not use collective operations will not require collective recovery.

The behavior of several MPI procedures will be different from those provided by non-fault tolerant implementations. For example, MPI_ABORT will only terminate the specified communicator. An MPI process rank within a communicator must not change, and thus MPI_COMM_SIZE and MPI_COMM_RANK must return the values as determined upon communicator instantiation, regardless of whether or not the communicator has lost processes. This implies, among other things, that processes removed from further participation in the communicator as part of a fault tolerance recovery process will be included in the size, assigned rank MPI_PROC_NULL.

A correct fault tolerant implementation of MPI requires that all procedures complete in some bounded time, including those defined as blocking. This requires, among other things, the ability to distinguish between failure and late arrival. As with any MPI non-blocking procedure, non-blocking fault tolerance procedures must be completed, using any of the (appropriate) procedures, such as MPI_WAIT, MPI_TEST, MPI_CANCEL, or their respective variants.

Three key terms capture the fault tolerance capabilities of MPI as they relate to a user application. In order to maintain general execution, MPI may *recover* a failed process. However, that process is not available to the calling program until that program *restores* or *rejoins* the process to a communicator.

> *Rationale.* Recovery of a failed process may occur without user knowledge, while restoring or rejoining the process for participation in the application program requires user participation. Another view is that recover is an action, restore is a state, and rejoin is a trait. (*End of rationale.*)

0.2 Initializing Fault Tolerance Support

The fault tolerance policy of a communicator and its member processes by MPI is specified by attaching attributes to the communicator using MPI_COMM_SET_ATTR. The following attributes are supported:

MPI_COMM_RESTORE_STRATEGY Specifies the restoration policy for a given communicator. Options are:

- MPI_NO_RESTORE (default)
- MPI_RESTORE

MPI_RANK_RESTORE_STRATEGY Specifies the restoration policy for a set of ranks. Input is an integer array, the first element of which is the number of ranks, followed by the list of ranks. Options are:

- MPI_NO_RESTORE (default)
- MPI_RESTORE

MPI_RESTORE_THRESHOLD_COUNT The required minimum size of the recovered communicator. If MPI is unable to recover the communicator to this size, an error will be returned. The default is all processes.

MPI_RESTORE_THRESHOLD_PERCENT The required minimum size of the recovered communicator, as a percent of its original size. If MPI is unable to recover the communicator to this size, an error will be returned. The default is all processes.

> *Discussion.* Rainer: This is somewhat clunky (apart from rounding errors), and with MPI_RESTORE_THRESHOLD_COUNT all should be said and done. (*End of discussion.*)

MPI_PROC_RESTORE_POLICY The policy for recovering failed processes.

- MPI_RESTORE_ALL: Attempt to recover all failed processes.
- MPI_RESTORE_SOME: Attempt to recover as many failed processes as possible.
- MPI_RESTORE_NONE: Do not recover any failed processes. (Default)

MPI_RESTORE_FN User defined function to be called by MPI immediately prior to returning from a communicator recovery procedure, providing a means for user participation in the recovery process. The default is NULL.

MPI_RESTORE_ALL_FN User defined function to be called by MPI immediately prior to returning from a collective communicator recovery procedure, providing a means for user participation in the collective recovery process. The default is NULL.

> *Discussion.* Rainer: Why the distinction of collective and single-proc recovery callback functions? (*End of discussion.*)

MPI_ERROR_REPORTING_FN User defined function to be called by MPI immediately prior to returning from an MPI procedure that has encountered an error. Provides a means for application specific error reporting. The default is NULL.

MPI_ERROR_NOTIFICATION Specifies the communicator failure notification policy.

- MPI_LOCAL_NOTIFICATION: Notify only processes directly impacted by the failure.
- MPI_GLOBAL_NOTIFICATION: Notify all processes of the failure. (Default)

MPI_DISCARD_PENDING_MESSAGES Specifies what to do with outstanding communication when process failure occurs.

- MPI_DISCARD_FAILED_PROCS: Discard only traffic associated with the failed process.
- MPI_DISCARD_ALL: Discard all traffic associated with the communicator. (Default)

Rationale. The default settings are intended to produce the same behavior as would a non-fault tolerant MPI implementation. Therefore, the user must explicitly attach attributes to communicators in order to enable a fault tolerant capability. (*End of rationale.*)

0.3 Checking the State of a Communicator

A program may query for the state of a communicator.

MPI_COMM_VALIDATE(comm, failed_rank_count, failed_ranks, return_code)

IN	comm	communicator (handle)
OUT	failed_rank_count	number of failed ranks in communicator (integer)
OUT	failed_ranks	array of failed ranks (integer)
OUT	return_code	return error code(integer)

Validation of the communicator. Upon return, a value other than 0 for failed_rank_count indicates the number of failed ranks in the communicator. This is a collective call.

Advice to users. Collective operations require that all participants receive the same values. However, it must be well-understood that it uses a phased system such that the result of the call s based on the callers' states prior to the call or at the start of the call but with the understanding that the results are not guaranteed to be accurate at the return of the call. (*End of advice to users.*)

MPI_COMM_IVALIDATE(comm, failed_rank_count, failed_ranks, request, return_code)

IN	comm	communicator (handle)
OUT	failed_rank_count	number of failed ranks in communicator (integer)
OUT	failed_ranks	array of failed ranks (integer)
OUT	request	request (handle)
OUT	return_code	return error code(integer)

Start a nonblocking validation of the communicator. This is a collective operation. These calls allocate a communication request object and associate it with the request handle (the argument request). The request can be used later to query the status of the communication or wait for its completion. A nonblocking call indicates that the system may start writing data into the failed_rank_count, failed_ranks buffer. However, the calling process should not access any part of the receive buffer after a nonblocking operation is called, until the operation completes as indicated by the successful return form an appropriate completion call.

Rationale. MPI behaves as if all communicators are intact. A process will only recognize a problem when it attempts to engage to communicate with a process. (*End of rationale.*)

Discussion. Rainer: Per the *Rationale* Processes either are notified, or when they communicate with another process. So, what does MPI_COMM_VALIDATE tell us? Immediately after returning from the function, a process may have died, anyway, right? Not sure who inserted this note: Need to update the status object for returned information. (*End of discussion.*)

Here is the caution from MPI 2.2 wrt MPI_Irecv. Inserting here in case an analogous situation is an issue here?:

> *Advice to users.* To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association," and "Problem with Register Optimization" in Section 16.2.2 on pages 482 and 485. (*End of advice to users.*)

0.4 Restoring a Process or a Communicator

The MPI implementation may recognize and perhaps recover a lost process or communicator for its own purposes. However, the communicator or process is not available to the user program until a procedure restores the communicator to a valid state or the process joins a valid communicator.

MPI_COMM_RESTORABLE(comm_names, count, return_code)

OUT	comm_names	array of communicators that may be recovered (strings)
OUT	count	the number of communicators that may be recovered (integer)
OUT	return_code	return error code(integer)

Returns a list of communicators that have been assigned an attribute that permits restoration in case of a fault. By default, will restore MPI_COMM_WORLD within the local view as well as MPI_COMM_SELF and MPI_COMM_NULL. The (user defined) strings returned by the procedure may be used to rejoin the failed communicators.

> *Discussion.* rbarrett: How is this name assigned? Not in the current list of attributes. Also, why is it needed? Actually I no longer know what I'm questioning :) (*End of discussion.*)

MPI_COMM_REJOIN(comm_names, comm, return_code)

IN	comm_names	communicator name (string)
OUT	comm	communicator (handle)
OUT	return_code	return error code(integer)

The calling process is re-associated with the input communicator using local recovery properties. Upon return it may participate in point-to-point communication within the communicator, but unless and until the communicator is fully restored, collective communication involving the communicator will not be valid.

MPI_COMM_RESTORE(ranks_to_restore, ft_status, return_code)

IN	ranks_to_restore	array of ranks to restore
OUT	ft_status	ft_status object
OUT	return_code	return error code(integer)

The blocking version of the process restoration procedure. The union of the requests across all ranks will be restored. Setting an element to MPI_COMM_NULL signifies that the calling process does not require the restoration of any processes in the communicator.

Advice to implementors. It is the responsibility of the MPI implementation to ensure that only a single instance of a given process exists at a given point in time. It must ensure that requests to restart a healthy process or multiple requests to restart the same process do not result in an internally inconsistent MPI state. This procedure is called by a surviving process that detects process failure. It is local in scope, and thus restores local communications (point-to-point, one-sided, data-type creation, etc.), but not collective communications. (*End of advice to implementors.*)

0.4.1 Return Status

(See MPI_Irecv discussion in existing spec. I've inserted placeholders, e.g. X and Y for illustrative purposes only.)

If multiple requests are completed by a single MPI function (see Section 3.7.5 *[FIXME: Need latex link]*), a distinct error code may need to be returned for each request. The information is returned by the status argument of MPI_COMM_RESTORE. The type of status is MPI-defined. Status variables need to be explicitly allocated by the user, that is, they are not system objects. In C, status is a structure that contains N fields named $X, Y, \ldots$, and MPI_ERROR; the structure may contain additional fields. Thus, status.MPI_X, status.MPI_Y and status.MPI_ERROR contain the x, y, and error code, respectively, of the ... In Fortran, status is an array of INTEGERs of size MPI_FT_STATUS_SIZE. The constants MPI_X, MPI_Y and MPI_ERROR are the indices of the entries that store the source, tag and error fields. Thus, status(MPI_X), status(MPI_Y) and status(MPI_ERROR) contain, respectively, the x, y, and error code of the restoration procedure.

In general, message-passing calls do not modify the value of the error code field of status variables. This field may be updated only by the functions in Section 3.7.5 *[FIXME: Need latex link]* which return multiple statuses. The field is updated if and only if such function returns with an error code of MPI_ERR_IN_STATUS.

Rationale. The error field in status is not needed for calls that return only one status, such as MPI_WAIT, since that would only duplicate the information returned by the function itself. The current design avoids the additional overhead of setting it, in such cases. The field is needed for calls that return multiple statuses, since each request may have had a different failure.

The status argument may also return other information. However, this information is not directly available as a field of the status variable and a call to MPI_GET_COUNT is required to "decode" this information. (This could refer to an as yet undefined function for retrieving such information.) (*End of rationale.*)

MPI_COMM_RESTORE_ALL(ranks_to_restore, request, return_code)

IN	ranks_to_restore	array of ranks to restore
OUT	request	request object (handle)
OUT	return_code	return error code(integer)

The collective blocking version of the process restoration procedure.

MPI_COMM_IRESTORE(ranks_to_restore, request, return_code)

IN	ranks_to_restore	array of ranks to restore
OUT	request	request object (handle)
OUT	return_code	return error code(integer)

The non-blocking version of the process restoration procedure.

MPI_COMM _IRESTORE_ALL(ranks_to_restore, request, return_code)

IN	ranks_to_restore	array of ranks to restore
OUT	request	request object (handle)
OUT	return_code	return error code(integer)

The collective non-blocking version of the process restoration procedure.

MPI_COMM_PROC_GEN(comm, generation, return_code)

IN	comm	communicator (handle)
OUT	generation	process generation (integer)
OUT	return_code	return error code (integer)

If restored by MPI fault tolerance capabilities, the process generation is incremented by one. (The initial generation is zero.)

MPI_COMM_GEN(comm, generation, return_code)

IN	comm	communicator (handle)
OUT	generation	communicator generation (integer)
OUT	return_code	return error code (integer)

If restored by MPI fault tolerance capabilities, the communicator generation is incremented by one. (The initial generation is zero.)

Discussion. Rainer: The following section must probably properly integrated... (*End of discussion.*)

0.4.2 Who's on first and what's on second

Text copied from status var discussion. I don't know what to call the structs, so I've inserted placeholders as defined by Abbott and Costello.

For convenience, a rank and its communicator are coupled in two MPI defined variables used in the procedures in this section. These variables are not system objects, and therefore must be explicitly allocated by the user.

In C, Whos_on_first is a structure that contains two fields named MPI_COMM and MPI_RANK. Thus, Whos_on_first.MPI_RANK and Whos_on_first.MPI_COMM contain the rank and its communicator, respectively. The structure may contain additional fields for internal use by MPI, but the user should ignore them since they are not standardized.

In Fortran, Whos_on_first is an array of INTEGERs of size MPI_XYZ_SIZE. The constants MPI_COMM and MPI_RANK are the indices of the entries that store the communicator and rank fields. Thus, Whos_on_first(MPI_RANK) and Whos_on_first(MPI_COMM) contain the rank and its communicator, respectively.

The result of the procedure is also reported using an MPI variable, named Whats_on_second. The fields are the same as Whos_on_first, with the addition of return_code, used to report the status of the procedure with regard to a specific rank in a communicator.

Rationale. Programmer convenience. (*End of rationale.*)

0.5 Callback functions

These callback functions provide the user a means for supplementing MPI recovery actions.

void(*MPI_COMM_ERROR_REPORT_FN) (comm, error_code, data)

IN	comm	communicator (handle)
IN	error_code	Error code as returned by failing rouine (integer)
IN	data	error description (void *)

Mechanism for increased error reporting. Error codes include:

- MPI_ERROR_RESTORED: As part of the recovery process, the library will invoke the local recovery function set by the MPI application at communicator creation. Only process local work, MPI and other, will be done within this user defined recovery function. In addition, the MPI library will discard any outstanding communication with the failed process, and reinitialize communications with the newly restored ranks. MPI_WAIT and MPI_TEST: calls made on MPI_REQUEST objects associated with the restored process and that were initialized before recovery will return MPI_ERROR_RESTORED, with the request object reset to MPI_REQUEST_NULL.

    Discussion. rbarrett: I don't understand this description as it related to this parameter. It sounds more like a general description of what happens during recovery: (*End of discussion.*)

- MPI_ERROR_PROC_FAILED: Indicates process failure. Returned information includes the communicator and rank information. Local error notification returns only those ranks associated with the affected communicator. Global error notification returns information regarding all communicators for which the rank is a member.

  Discussion. rbarrett: Don't know what this means: Each failed process will be reported with all communicators in use, giving it's appropriate rank within each such communicator. (*End of discussion.*)

Discussion. Rainer: In the above, what is e.g. the error_code being passed? Is it only these two values? I'd rather think, these are two new error values being specified in the possible list of error-values. (*End of discussion.*)

void(*MPI_COMM_RESTORE_FN) (comm)

IN	comm	communicator (handle)

Invoked prior to return from MPI_COMM_RESTORE or MPI_COMM_IRESTORE.

void(*MPI_COMM_RESTORE_ALL_FN) (comm)

IN	comm	communicator (handle)

Collective version of MPI_COMM_RESTORE_FN. That is, called prior to return from MPI_COMM_RESTORE_ALL or MPI_COMM_IRESTORE_ALL.

Advice to users. This callback function is invoked by the MPI library right before the library returns from the local repair functions. This provides the application with a way to invoke communicator specific code on recovery, supporting layered library recovery. (*End of advice to users.*)

0.6 Supplement

This section contains information that may be relevant across other areas of the full MPI specification in order to effectively support the fault tolerance capabilities under consideration.

0.6.1 The Dynamic Process Model

Section 10.2 of the MPI 2.2 specification[1] describes the mechanism that allows the user to create and terminate processes within a communicator. It includes strong cautions regarding the runtime environment, and includes an example of the issues associated with a batch queueing system. For example, the 2.2 spec states, *"It provides a mechanism to establish communication between the newly created processes and the existing MPI application."* The FT text should adhere to this style of description since FT adheres to this concept. e.g. "newly created" and "recovered" and "restored" are related ideas.

I'm placing the "Advice to Users" here for convenience, but it will move to the FT section. We should ensure a smooth relationship between these two ideas (i.e. dynamic processes and FT).

> *Advice to users.* An MPI application configured for fault tolerance must include mechanisms for operating within the context of its specific runtime environment. For example... (*End of advice to users.*)

But we must still maintain a separation with the dynamic process section in order to avoid confusion. That is, FT is responsible for spawning new processes whereas dynamic processes gives this responsibility to the user. Hmmm, sounds like another "Advice to Users" blurb.

0.6.2 PVM discussion

The examples in this section are general, referring to a "batch queueing system", "Network of workstations managed by a load balancing system", and a "large SMP with Unix". The exception to this generality is "Network of workstations with PVM". Yet the current text includes a discussion of the issues associated with such a runtime system, e.g. "It does not provide "operating system" services, such as a general ability to query what processes are running, to kill arbitrary processes,to find out properties of the runtime environment (how many processors, how much memory, etc.)."

At the risk of incurring the wrath of PVM fans (of which I am one), I will ask: Should this direct reference be eliminated from the MPI 3 spec? I would argue yes, but am certainly open to alternative views. (Further evidence for elimination: Condor is not called out under "managed by a batch queueing system", yet Condor may be more in use than PVM (and forms the basis of load leveler?) Stop digging, Barrett...

This then leads into a more general notion of the PVM-like functionality within the spec, and for example should be considered within the context of other areas, such as the buffered send. Is this in use?

0.6.3 Fortran Issues

Throughout the document I notice qualifiers with regard to Fortran. For example, in the MPI_Irecv section:

Advice to users. To prevent problems with the argument copying and register optimization done by Fortran compilers, please note the hints in subsections "Problems Due to Data Copying and Sequence Association" and "Problem with Register Optimization" in Section 16.2.2 on pages 482 and 485. (End of advice to users.)

No doubt intruding in the business of that working group, but if it affects MPI_Irecv *is probably affects FT, e.g.* MPI_Comm_ivalidate*. Just a heads up.*

0.6.4 General

Discussion of request handle *p52, MPI 2.2: "The request can be used later to query the status of the communication or wait for its completion." This is in the discussion of the non-blocking receive, but it applies to any use of a request handle. Shouldn't this be a stronger statement? That is, the request* can *be used to query, but*must *be used to complete the operation. I'll bet this has been discussed, so presume there is a good* *reason for the language. But it also seems to me that failure to complete a request is a "bug" in an application.*

Bibliography

[1] *MPI Forum. MPI : A Message Passing Interface Standard, version 2.2.* `http://www.mpi-forum.org/docs/mpi22-report.pdf`, *2009.* 0.6.1

[2] *Richard L. Graham, Richard Barrett, Greg Bronevetsky, Erez Haba, Gregory Koenig, Hideyuki Jitsumoto, Thomas Herault, Joshua Hursey, Adam Moody, Kannan Narasimhan, Howard Pritchard, and David Solt. Creating Fault Tolerant Applications Using MPI.* In preparation. 0.1

[3] *Richard L. Graham, Richard Barrett, Greg Bronevetsky, Erez Haba, Gregory Koenig, Hideyuki Jitsumoto, Thomas Herault, Joshua Hursey, Adam Moody, Kannan Narasimhan, Howard Pritchard, and David Solt. Towards support for fault tolerance in the mpi standard.* In preparation. 0.1