Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems

Graham E. Fagg, Edgar Gabriel, George Bosilca,
Thara Angskun, Zhizhong Chen, Jelena Pjesivac-Grbovic, Kevin London
and Jack J. Dongarra

Innovative Computing Laboratory,
Department of Computer Science, 1122 Volunteer Blvd., Suite 413,
University of Tennessee, Knoxville, TN-37996, USA.

{fagg, egabriel, bosilca, angskun, zchen, pjesa, london, dongarra}@cs.utk.edu

1. Motivation

1.1 Trends in High Performance Computing

End-users and application developers of high performance computing systems have today access to larger machines and more processors than ever. Systems such as the Earth Simulator, the ASCI-Q machines or the IBM Blue Gene consist of thousands or even tens of thousand of processors. Machines comprising 100,000 processors are expected for the next years.

A critical issue of systems consisting of such large numbers of processors is the ability of the machine to deal with process failures. Concluding from the current experiences on the top-end machines, a 100,000-processor machine will experience a process failure every few minutes[1]. While on earlier massively parallel processing systems (MPPs) crashing nodes often lead to a crash of the whole system, current architectures are more robust. Typically, the applications utilizing the failed processor will have to abort, the machine, as an entity is however not affected by the failure. This robustness has been the result of improvements at the hardware as well as on the level of system software.

1.2 Current Parallel Programming Paradigms

Current parallel programming paradigms for high-performance computing systems are mainly relying on message passing, especially on the Message-Passing Interface (MPI) [12][13] specification. Shared memory concepts (e.g. OpenMP) or parallel programming languages (e.g. UPC, CoArrayFortran) offer a simpler programming paradigm for applications in parallel environments, however they either lack the scalability to tens of thousands of processors, or do not offer a feasible framework for complex, irregular applications. The message-passing paradigm on the other hand provides a mean to write highly scalable algorithms, abstracting and hiding many architectural decisions from the application developers.

MPI in its current specification is however not dealing with the situation mentioned above, where one or more processes are becoming unavailable during runtime. Currently, MPI gives the user the choice between two possibilities of how to handle failures. The first one, which is also the default mode of MPI, is to immediately abort the application. The second possibility is just slightly more flexible, handing the control back to the user application without guaranteeing however, that any further communication can occur. The latter mode has mainly the purpose to give an application the possibility to perform local operations before exiting, e.g. closing all files or writing a local checkpoint.

1.3 Recent developments in MPI

The MPI user and developer community is currently going through dynamic changes. More and more MPI implementations are meanwhile available, each focusing on different aspects. The manufacturers of high-performance computing systems focus on improving the performance of MPI-1 functions as well as on implementing more and more of MPI-2. The two most widespread, freely available MPI libraries (MPICH[21], LAM/MPI[9]) are both coming closer to support the full MPI-2 specification.

Many MPI implementations are dealing with aspects of distributed and Grid computing, focusing on the given hierarchies, security aspects as well as user-friendliness. Examples are PACX-MPI[3], MPICH-G2[14], Stampi[19], MetaMPICH or GridMPI[8].

Another set of projects is dealing with various aspects of fault-tolerance, e.g. MPI/FT[15], MPI-FT[17], MPICH-V[5] or LA-MPI[16]. Many of these projects are providing a checkpoint-restart interface, allowing an application to restart from the last consistent checkpoint in case an error occurs.

1.4 The need for extending the MPI specification

Summarizing the findings of the introduction so far, there is a mismatch between the capabilities of current high performance computing systems and the mainly used parallel programming paradigm. While the machines are getting more and more robust (hardware, network, operating systems, file systems) the MPI specification does not leave room for fully exploiting the capabilities of the current architectures. Checkpoint/restart, the only currently available option, does have its performance and conceptual limitations, if machines with tens of thousand of processors are considered. In fact, one of the main reasons for many research groups to stick still to the PVM[4] communication library instead of switching to MPI is the capability of the first one to handle process failures.

> If today's and tomorrows high performance computing resources shall be used as a means to perform single, large scale simulations and not solemnly as a platform for high throughput computing, extending the main communication library of HPC systems, respectively the main programming paradigm, to deal with aspects of fault-tolerance is inevitable.

Therefore, we present in this document the results of work conducted during the last four years, which produced:

- A specification called 'Proposal for Extensions to the Message-Passing Interface for Process Fault-Tolerance'
- An implementation of this specification
- Numerous application scenarios showing the feasibility of the specification for scientific, high performance computing.

The rest of the document is organized as follows: Section 2 presents a summary of the specification for a Fault-Tolerant MPI (FT-MPI). The complete specification is attached as Appendix A. In section 3 we present some of the architectural decisions on implementing the FT-MPI specification. Section 4 presents a wide variety of application scenarios using the FT-MPI specification, ranging from dense linear algebra examples to parallel equation solvers and a master-slave code. Finally, section 5 summarizes the paper and presents the ongoing work.

2. Extensions to the Message-Passing Interface for Process Fault –Tolerance

This section summarizes the FT-MPI specification. The full document can be found in Appendix A. Handling fault-tolerance typically consists of three steps: failure detection, notification and recovery. The FT-MPI specification makes no general assumptions about the first two issues, with the exception that it assumes, that the run-time environment discovers failing processes and all processes of the according parallel jobs are notified about the failure events.

The notification of failed processes is passed to the MPI application through the usage of a special error code. As soon as an application process has received the notification of a death event through this error code, its general state is changing from ***no failures*** to ***failure recognized.*** While in this state, the process is just allowed to execute certain actions. These actions are depending on various parameters and are detailed later in the document.

The recovery procedure is considered to consist of two steps again: recovering the MPI library and the run-time environment, and recovering the application. The latter one is considered to be the responsibility of the application.

The main problems the FT-MPI specification is dealing with are answers to the following questions:

1. What are the necessary steps and options to start the recovery procedure and therefore change the state of the processes back to ***no failure***?
2. What is the status of the MPI objects after recovery?
3. What is the status of ongoing communication and messages during and after recovery?

The first question is handled by the so-called ***recovery mode***, the second by the ***communicator mode***, the third by the ***message mode*** respectively the ***collective communication mode***.

The recovery mode defines how the recovery procedure can be started. Currently, there are three options defined:

- an automatic recovery mode, where the recovery procedure is started automatically by the MPI library as soon as a failure event has been recognized
- a manual recovery mode, where the application has to start the recovery procedure through the usage of a special MPI function
- a recovery mode, where the recovery procedure does not have to be initiated at all. However, any communication to failed processes will raise an error.

The status of MPI objects after the recovery operation is depending on whether they contain some global information or not. As for MPI-1, the only objects containing global information are groups and communicators. These objects are 'destroyed' during the recovery procedure and only the objects available after MPI_Init are re-instantiated by the library (MPI_COMM_WORLD and MPI_COMM_SELF).

Communicators and group can have different formats after recovery operation. Failed processes can either be replaced (FTMPI_COMM_MODE_REBUILD), or not. In case the failed processes are not replaced, the user still has two choices: the position of the failed process can be left empty in groups and communicators (FTMPI_COMM_MODE_BLANK) or the groups and communicators can shrink such that no gap is left (FTMPI_COMM_MODE_SHRINK). For both modes a precise description of all MPI-1 functions are given in Appendix A.

Furthermore, the specification has to clarify what the status of currently ongoing messages is while an error occurs and is recognized. In one mode, all currently ongoing messages are cancelled by the system. This mode is mainly useful for applications, which on an error roll-back to the last consistent state in the application. As an example, if an error occurs in iteration 423 and the last consistent state of the application is from iteration 400, than all ongoing messages from iteration 423 would just confuse the application after having performed the roll-back. The

second mode completes all ongoing messages after the recovery operation, with the exception of the messages to and from the failed processes. This mode requires, that the application keeps precisely track of the state of each process, minimizing the roll-back procedure. Similar modes are available for collective operations, which can either be executed in an atomic or a non-atomic mode.

3. An Implementation of the FT-MPI specification

The University of Tennessee implemented the specification presented in the previous section. The current implementation is relying on the HARNESS[2][7] framework. HARNESS (Heterogeneous Adaptable Reconfigurable Networked SyStems) provides a fault-tolerant, dynamic run-time environment, which is used by FT-MPI for process management and failure notification.

UTK's implementation of the FT-MPI specification proves, that the specification is not just a theoretical framework, but that it is practically working.

The currently available functionality includes the full MPI-1.2 specification, as well as several sections of the MPI-2 document. Lots of efforts have been furthermore invested in optimizing the collective operations and the derived datatype section of MPI. A multi-protocol device supporting besides TCP/IP also various other protocols (e.g. shared memory, myrinet) is currently in the testing phase.

UTK's implementation has proven in various benchmarks and application scenarios, that the performance of FT-MPI is comparable to the current state-of-the art MPI libraries as long as no error occurs. This indicates, that the new specification does not harm the performance of applications in case no error occurs. The results furthermore show, that the FT-MPI specification is compatible to the current specifications of MPI, since all current MPI applications work without any modifications with FT-MPI.

Currently ongoing work is also the abstraction of the features needed for implementing the fault-tolerant features into an abstract device interface (FT-ADI). Thus, different run-time environments could be used to implement a specification of FT-MPI, e.g. a simple environment relying on shared files for processing having a common file-system. UTK's FT-MPI implementation is available for free download at http://icl.cs.utk.edu/ftmpi/.

4. Usage scenario

Simultaneously to the development of the specification and UTKs implementation of FT-MPI, a large set of applications have been tested and benchmarked. Most of these rely on a technique called in-memory checkpoint. This technique avoids writing checkpoint files by distributing additional information based on encoding techniques like the Reed-Solomon Algorithm on other processes. In case an error occurs, the application need not be restarted, but the additional information is used to reconstruct the data of the failed process. Especially for large numbers of processes, this technique improves the performance of the application dramatically compared to writing and reading checkpoint files, since it avoids typically slow file operations. Among the applications using this technique are

- A parallel, preconditioned conjugate gradient solver[6],
- A dense matrix multiplication,
- LU factorization,
- QR factorization and

- A parallel spectral transform Shallow Water Code (PSTSWM)

These applications show, that using the FT-MPI specification one can significantly improve the performance of the application in case an error occurs. As with most fault-tolerant applications known in the literature, there is however a trade-off between the additional resources used to achieve fault-tolerance (memory, processes) and the level of fault-tolerance (e.g. number of process failures which can be survived by the application).

The in-memory checkpointing technology is a very promising approach even for complex applications, as shown for the parallel spectral transform shallow water code (PSTSWM). The reason for the applicability of this technique to complex applications is, that most real-world simulations have anyway a checkpoint-restart interface built in. To use an in-memory checkpoint algorithm usually just requires a modification of the checkpoint and restart routines and not of the whole application.

A fault-tolerant manager-worker framework has been furthermore developed, which does not use in-memory checkpointing[6]. The key point of this framework is to show, that all applications, which can be written using a master-slave paradigm (e.g. parameter sweep studies) can easily be adapted to FT-MPI. The current implementation of the framework can make use of all three communicator modes (rebuild, blank and shrink).

Recent work by Geist and Engelman[1] present new algorithms for solving partial differential equations, which are called ‘naturally fault tolerant algorithms’. Based on mesh-less methods and chaotic relaxation, Geist and Engelman show, that the algorithm still converges correctly, as long as a marginal number of processes are failing, which do not have to be replaced (e.g. using the blank mode). A marginal number of processes in this context can still be 100 process failures in a 100,000-processor job.

The specification has proven to be powerful enough to support not just one of a kind applications, but to support various approaches to handle fault-tolerance and leave room for users to handle fault-tolerance according to the requirements of their applications.

5. Summary

We have presented in this paper an extension to the MPI specification for handling process fault tolerance. Together with the specification, the FT-MPI team at the Innovative Computing Laboratory of the University of Tennessee has developed an implementation of the specification and various application scenarios.

The current specification is in the spirit of the MPI-1 and MPI-2 documents: similarly to MPI-1 and MPI-2, which do not restrict the application developers by offering different data decomposition techniques, FT-MPI does not specify how to handle fault-tolerance on the application level. Instead, FT-MPI offers a rich set of techniques for failing MPI processes and defines the status of MPI objects in case a failure occurs, leaving the applications room for implementing their preferred way to handle fault-tolerance.

Acknowledgments
This material is based upon work supported by the Department of Energy under Contract No. DE-FG02-02ER25536. The NSF CISE Research Infrastructure program EIA-9972889 supported the infrastructure used in this work.

6. References

[1] Al Geist and Christian Engelmann: Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors, Journal of Parallel and Distributed Computing, to be published.

[2] Beck, Dongarra, Fagg, Geist, Gray, Kohl, Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. Scott, V. Sunderam, "HARNESS: a next generation distributed virtual machine", Journal of Future Generation Computer Systems, (15), Elsevier Science B.V., 1999.

[3] Edgar Gabriel, Michael Resch, and Roland Ruehle, Implementing MPI with Optimized Algorithms for Metacomputing, in Anthony Skjellum, Purushotham V. Bangalore, Yoginder S. Dandass, 'Proceedings of the Third MPI Developer's and User's Conference', MPI Software Technology Press, Starkville, Mississippi, 1999.

[4] G. Geist, J. Kohl, R. Manchel, and P. Papadopolous, New Features of PVM 3.4 and Beyond, PVM Euro User's Group Meeting, pp. 1-10, September, 1995.

[5] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, Anton Selikhov, "MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes", *In Proceedings of SuperComputing 2002. IEEE, Nov.*,2002.

[6] Graham E. Fagg, Edgar Gabriel, Zhizhong Chen, Thara Angskun, George Bosilca, Antonin Bukovsky and Jack J. Dongarra: 'Fault Tolerant Communication Library and Applications for High Performance Computing', LACSI Symposium 2003, Santa Fe, October 27-29, 2003.

[7] Graham Fagg, Antonin Bukovsky, and Jack Dongarra, HARNESS and Fault Tolerant MPI, Parallel Computing, Volume 27, Number 11, pp 1479-1496, October 2001, ISSN 0167-8191

[8] GridMPI: http://www.gridmpi.org,

[9] Jeffrey M. Squyres and Andrew Lumsdain, 'A Component Architecture for LAM/MPI, in in Jack J. Dongarra, Domenico Laforenza, Salvatore Orlande (Eds.), 'Recent Advances in Parallel Virtual Machine and Message Passing Interface', Lecture Notes in Computer Science vol. 2840, 2003.

[10] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. MPI-The Complete Reference. Volume 1, The MPI Core, second edition (1998).

[11] Message Passing Interface Fofum:'MPI-2 Journal of Development', http://www.mpi-forum.org, 1997.

[12] Message Passing Interface Forum:'MPI: A Message Passing Interface Standard', http://www.mpi-forum.org, 1995.

[13] Message Passing Interface Forum:'MPI-2: Extensions to the Message Passing Interface Standard', http://www.mpi-forum.org, 1997

[14] N. Karonis, B. Toonen, and I. Foster, MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface, Journal of Parallel and Distributed Computing, to appear 2003.

[15] R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjellum, Y. Dandass, and M. Apte, MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing, in Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid, held in Melbourne, Australia, 2001.

[16] Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger and Mitchel W. Sukalski, A Network-Failure-Tolerant Message-Passing System For Terascale Clusters, ICS02, June 22-26, 2002, New York, New York, USA.

[17] Soulla Louca, Neophytos Neophytou, Adrianos Lachanas, Paraskevas Evripidou, "MPI-FT: A portable fault tolerance scheme for MPI", Proc. of PDPTA '98 International Conference, Las Vegas, Nevada 1998.

[18] T. Bemmerl: MetaMPICH: Flexible Coupling of Heterogeneous MPI Systems}. http://www.lfbs.rwth-aachen.de/~martin/MetaMPICH/metaframe.html August 2001.

[19] T. Imamura, Y. Tsujita, H. Koide, H. Takemiya, An Architecture of Stampi: MPI library on a cluster of parallel computers, in J. Dongarra, P. Kacsuk, N. Podhorszki (Eds.) ' Recent Advances in Parallel Virtual Machine and Message Passing Interface', Lecture Notes in Computer Science vol. 1908, pp. 200-207, Springer, Berlin 2000.
[20] T. Kielmann, R.F.H. Hofman, H.E. Bal, MagPIe: MPI's collective communication operations for clustered wide area systems, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'99), pp.131-140, ACM, 1999.
[21] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, 22(6):789-828, September 1996.
[22] William Gropp and Ewing Lusk, Fault Tolerance in MPI Programs, to appear in Journal of High Performance Computing and Applications, 2003.

Appendix A
FT-MPI: Proposal for Extentions to the Message-Passing Interface for Process Fault-Tolerance

February 16, 2004

Innovative Computing Laboratory,
Computer Science Department,
University of Tennessee, Knoxville

Abstract

This document describes extentions to the MPI-1.2 and MPI-2 standards for introducing process fault-tolerance in MPI.

Contents

1 Introduction **2**
1.1 Background 2
1.2 Organization of this Document 3

2 Basic fault-tolerance issues **4**

3 Recovery Modes **7**
3.1 Pathological failures 8

4 Communicator modes **9**
4.1 FTMPI_COMM_MODE_BLANK 11
4.2 FTMPI_COMM_MODE_SHRINK 13

5 Message modes **14**
5.1 Non-deterministic communication in MPI 16
5.2 Collective operations 18

6 Miscellany **20**
6.1 New Attributes 20
6.2 New return code for MPI_Init 21
6.3 Fault tolerance and error-handlers 21

7 Conclusions **23**

Chapter 1

Introduction

1.1 Background

Application developers and end-users of high performance computing systems have today access to larger machines and more processors than ever before. High-end systems consist nowadays of thousands of processors. Additionally, not only the individual machines are getting bigger, but with the recently increased network capacities, users have access to higher number of machines and computing resources. Concurrently using several computing resources, often referred to as Grid- or Metacomputing, further increases the number of processors used in each single job as well as the overall number of jobs, which a user can launch.

With increasing number of processors however, the probability, that an application is facing a node or link failure is also increasing. While on earlier massively parallel processing systems (MPPs), a crashing node often was identical to a system crash, current systems are more robust. Usually, the application running on this node has to abort, however, the system in general is not effected by a processor failure. In Grid environments, a system may additionally become unavailable for a certain time due to network problems, leading to a similar problem from the application point of view like a crashing node on a single system.

The Message Passing Interface (MPI) [1, 2] is the de-facto standard for the communication in scientific applications. However, MPI in its current specification gives the user no possibility to handle the situation mentioned above, where one or more processors are becoming unavailable during runtime. Current MPI specifications give the user the choice between two possibilities of how to handle a failure. The first possibility is the default mode,

which is to immediately abort the application. The second possibility is to hand the control back to the user application (if possible) without guaranteeing, that any further communication can occur. The latter mode mainly has the purpose of giving the application the possibility to close all files properly, write maybe a per-process based checkpoint etc., before exiting the application.

The goal of this document is to bridge the gap between the more and more robust, fault-tolerant hardware which has evolved over the last years and the main programing paradigm used by scientific application, which does not offer process fault-tolerance in its current specifications.

1.2 Organization of this Document

This document is organized as follows: in chapter 2 we introduce the basic terminologies and definitions used throughout the document. The following chapters 3, 4, 5 specify the different failure recovery models supported by FT-MPI. Chapter 6 defines various other minor improvements, which help writing fault-tolerant applications using the FT-MPI specification.

Chapter 2

Basic fault-tolerance issues

Fault tolerance usually covers three steps:

- Fault detection
- Notification
- Recovery

Fault detection is the process of discovering that one or several processes have failed. While the FT-MPI specification makes no statement about **how** faulty processes are discovered, it assumes **that** they are discovered by the run-time environment. FT-MPI makes no assumption about **when** faulty processes are discovered. FT-MPI does furthermore not specify when a process is considered to have failed.

Notification deals with the problem of how the other MPI processes of parallel job get informed about the failure event. FT-MPI makes no assumptions **when** the processes are notified nor does it assume, that all processes are notified simultaniously. FT-MPI just specifies, that all processes of a parallel job are receiving a notification about death events.

The notification of failed processes are passed to the MPI application through a special error code. For achieving the largest possible conformance to the MPI-1 and MPI-2 specification, FT-MPI is not introducing a new error code, but defines, that MPI_ERR_OTHER is just to be used to signal the MPI application, that some processes have unexpectedly left the run-time environment.

As soon as an application process has received the notification of a death event through the MPI error code MPI_ERR_OTHER, its general state has changed from 'NO FAILURES' to 'FAILURE RECOGNIZED'. While in this

state, the process is just allowed to execute certain actions. These actions are depending on various other parameters and are detailed later in the document.

> *Rationale:* While the introduction of a new error-code indicating failed processes would have been desirable, currently we consider the requirement to have an FT-MPI specification, which allows to run an application written according to the FT-MPI specification on any regular non FT-MPI conformant implementation of MPI as more important than a 'cleaner' solution at this point. It is however still desirable to introduce a separate error-code in future specifications. A future of FT-MPI will furthermore deal with the problem of whom to notify in dynamic MPI-2 environments.

> *Advice to implementors:* A high quality implementation of the FT-MPI specification shall distinguish to the largest possible extent, whether a process has died due to an error in the application (e.g. segmentation violation) or because of a failure in the hardware or run-time environment.

The recovery procedure is the superset of steps necessary to move the status of MPI application processes and the MPI run-time environment from 'FAILURE RECOGNIZED' back to 'NO FAILURE'. Most of the FT-MPI specification is dealing with the problem how to move processes back into the 'NO FAILURE' mode, and what options are given to the user.

The recovery procedure is considered to have two steps:

1. Recovering the MPI run-time environment and the MPI library. This step will be handled in great details in the following sections.

2. Recover the application and application data: this step is considered to be the responsibility of the application and **not** of the MPI library. The FT-MPI specification makes no assumptions or statements about how an application recovers data from one or several lost processes.

> *Rationale:* in contrary to many currently available projects, FT-MPI does not provide an interface for checkpointing and recovering user data. Such an interface might be added in later versions of the FT-MPI specification, is however not considered in the current version.

The FT-MPI specification gives answers to the following questions related to the recovery process:

1. What are the required steps and/or options to start the recovery procedure once the processes are in the 'FAILURE RECOGNIZED' status?

2. What is the status of MPI objects and processes after recovery?

3. What is the status of ongoing communication (point-to-point communication as well as collective operations) after recovering from a failure?

The first question is handled by the **recovery mode** (FTMPI_RECOVERY_MODE), the second by the **communicator mode**(FTMPI_COMM_MODE) and the third by the **message mode**(FTMPI_MSG_MODE) respectiveley the **collective communication mode** (FTMPI_COLL_MODE).

Chapter 3

Recovery Modes

The user has three possibilities how the recovery procedure can be started:

1. FTMPI_RECOVERY_MODE_AUTO: as soon as the MPI library realizes, that a death event has occured, it automatically starts the recovery process. No interaction from the application is required. After the recovery has been successfully finished, the error handler of MPI_COMM_WORLD is called, since other communicators are not available after recovery. The state of communicators, groups and other objects are defined in later sections.

2. FTMPI_RECOVERY_MODE_MANUAL: like on any other error, the MPI library calls the error handler attached to the current communicator. The user is however not allowed to call any MPI function involving communication before the recovery has been started.

   To start the recovery, the user has to call MPI_Comm_dup on MPI_COMM_WORLD. The input argument of MPI_Comm_dup should be MPI_COMM_WORLD, the output argument is undefined and should be ignored by the application.

   ```
   oldcomm = MPI_COMM_WORLD;
   MPI_Comm_dup ( oldcomm, &newcomm );
   ```

   Rationale: The semantics chosen to initiate the recovery procedure manually has once again been driven by the desire to avoid introducing a new MPI function. Introducing a separate function in later version to avoid the dual functionality of MPI_Comm_dup is highly recommended.

3. FTMPI_RECOVERY_MODE_IGNORE: in this mode, the recovery procedure does not have to be initiated at all, as long as no communication with the dead processes are required. Communication involving dead processes (point-to-point operations, collective operations as well as communicator creations) will raise an error and will not be executed.

   Rationale: This mode has been designed with two things in mind. First, since the recovery procedure is a collective operation, it can be desireable to avoid this collective operation for large numbers of processors (e.g. 100,000). Second, there is a class of applications often refered to as 'naturaly fault-tolerant' which do not require any special handling on the application level to deal with failed processes.

3.1 Pathological failures

An MPI library can still abort if a pathological failure has occured from which it can not recovery. Typical reasons for pathological failures could be:

- All processes of an MPI job have failed before a recovery operation could be started.
- The MPI library has no 'room left' where to respawn processes.

Chapter 4

Communicator modes

This section defines the status of MPI processes and objects after a recovery operation. As a rule of thumb, all MPI objects containing non local information are destroyed and have to be re-established by the application. The following objects do **not** contain non-local information and will be therefore available on surviving processes after recovery:

- Datatypes (MPI_Datatype)
- Operations (MPI_Op)
- Error handlers (MPI_Errhandler)
- Info objects (MPI_Info, MPI-2)

The following list shows the objects which are destroyed during the recovery procedure:

- Groups (MPI_Group)
- Communicators (MPI_Comm)
- Windows (MPI_Win, MPI-2)
- Files (MPI_File, MPI-2)

Requests are in this context 'special' object, their behaviour is depending on the message mode and is explained in section 5. Windows and Files will be handled in more details in later versions of the specification.

> *Rationale:* Although groups are considered to be local objects in MPI, they contain usually a list of participating processes. Since this list might have changed during recovery, all user defined groups are considered to be potentially out of date.

After the recovery operation, the user has access to the same non-local operations like after MPI_Init. These are:

- Groups: none
- Communicators: MPI_COMM_WORLD and MPI_COMM_SELF.

> *Rationale:* It would be theoretically possible to modify non-local objects on the surviving processes such, that they contain the up-to-date information of the run-time environment. However, assuming that failed processes are replaced by the run-time environment (see the following section) there is no MPI function call to pass the additional handles to the re-spawned processes in a portable, MPI conforming manner.

Groups and Communicators can have different formats after the recovery procedure, depending on the communicator mode. The communicator mode specifies, how the run-time environment should treat failed processes. Four modes are currently defined:

1. FTMPI_COMM_MODE_ABORT: like in MPI-1 and MPI-2, the MPI library will abort the execution if one or several processes have failed. This mode is available for backward compatibility.
2. FTMPI_COMM_MODE_REBUILD: failed processes will be replaced by the run-time environment. Surviving processes will retain their rank in MPI_COMM_WORLD. No assumptions are made within the FT-MPI specification **where** the new processes are placed.
3. FTMPI_COMM_MODE_BLANK: failed processes will not be replaced, the size of MPI_COMM_WORLD will remain unchanged. However, the failed processes are blanked out and treated similarly to MPI_PROC_NULL. Detailed specifications about operations using blank processes can be found in the next subsections.
4. FTMPI_COMM_MODE_SHRINK: failed processes will not be replaced. The size of MPI_COMM_WORLD will be adjusted to the number of

surviving processes. This includes also, that the ranks of some processes in MPI_COMM_WORLD will change. FT-MPI requires that the sequence of surviving processes is identical before and after recovery. Figure 4.1 is showing an example, where two out of four processes fail, how the ranks are assigned after recovery.

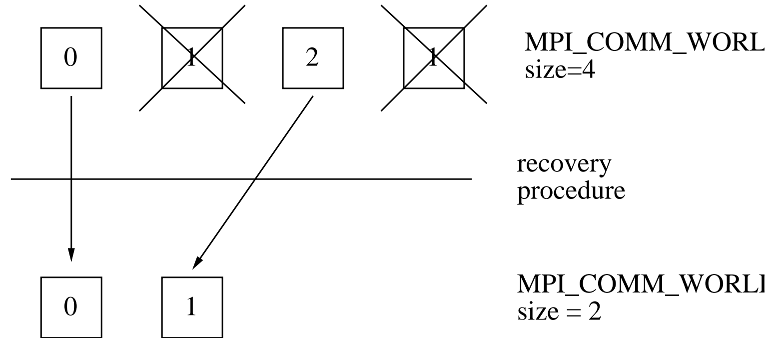

Figure 4.1: Example of the communicator mode FTMPI_COMM_MODE_SHRINK.

FTMPI_COMM_MODE_ABORT and FTMPI_COMM_MODE_REBUILD require no changes to the MPI-1 and MPI-2 specification after recovery. The communicator modes FTMPI_COMM_MODE_BLANK and FTMPI-_COMM_MODE_SHRINK introduce some new aspects to MPI and are detailed in the following.

4.1 FTMPI_COMM_MODE_BLANK

Point-to-point operations A blank process is defined to behave like MPI_PROC_NULL in the MPI-1 specification. This includes, that sending a message to a blank process will not raise an error, however no data is transmitted. Receive operations from a blank processes will return a null-status (see section 3.11 in MPI-1), the receiver buffer is unchanged.

Collective operations For collective operations, two different issues have to be taken into account for the blank mode. If the root of one of the rooted collective operations (MPI_Bcast, MPI_Reduce, MPI_Gather(v), MPI_Scatter(v)) is a blank process all processes will return immediatly. No input or output buffer is modified.

If a non-root process of the same operations is blank, this process will not contribute to the result of the collective operation. This means especially:

- Bcast: no data will be sent to the blank process(es).
- Reduce: blank processes do not contribute to the global result. Special care has to be taken not to assume predefined values for the blank processes, since this could alter the result (e.g. using zero for a blank process in MIN, MAX or PROD operations). It is invalid to return a blank process as the result of a reduce operation using MAXLOC and MINLOC. User defined operations will not be called for blank processes.
- Gather(v): in the receive buffer of the root, the data segements assigned to the blank process(es) will be untouched.
- Scatter(v): no data will be sent to the blank process(es).

The rules for non-rooted operations can be directly derived from the rules for rooted operations. The implementation has to ensure, that for operations, which are implemented as a combination of other collective operations (e.g. MPI_Allreduce implemented as an MPI_Reduce followed by an MPI_Bcast) a temporary root node is chosen, which is not a blank process.

Group and Communicator creation functions All operations defined in MPI-1 and MPI-2 for deriving new groups and communicators are valid for blank processes aswell. This includes:

- it is valid to split communicators containing blank processes
- it is valid to derive a group from a communicator which contains blank processes
- it is valid to include/exclude a blank processes from a group
- it is valid to generate new communicators from groups containing or excluding blank processes.

The group and commmunicator comparison functions MPI_Group_compare and MPI_Comm_compare need not be able to distinguish between two blank processes.

All topology functions might include blank processes. The outcome of the topology functions returning the ranks of neighbor processes might be a blank process.

Rationale: it would be possible to modify the semantics of MPI_Cart_shift and MPI_Graph_neighbors such that they return the first non-blank process according to the user settings. However, this would be equal to ignoring the 'distance' argument provided by the user, or at least a redefinition of it.

4.2 FTMPI_COMM_MODE_SHRINK

In this communicator mode, the ranks of MPI processes before and after recovery might change, as well as the size of MPI_COMM_WORLD does change. The appealing part of this communicator mode however is, that all functions specified in MPI-1 and MPI-2 are still valid without any further modification, since groups and communicators do not have wholes and blank processes.

Chapter 5

Message modes

This section explaines the expected behavior of messages before, during and after recovery. The major problem arises from the fact, that typically some messages will be 'within the system' while an error occurs. In this section, we define the behavior of messages which are on the fly why an error occurs. Two general rules apply for all message modes:

1. All messages from and to dead processes are discarded, independent of recovery, communicator or message mode.

2. All collective operations will stop immediatly and all messages initiated by collective operations will be discarded, independent of the recovery, communicator or message mode. In the following subsecion, we will furthermore discuss the behavior of collective operations while an error occurs.

For explaining the difference between the two message modes provided by the FT-MPI specification, we would like to introduce the terminology of a *generation count* for communicators. If MPI_COMM_WORLD has a generation count of x before a process failes, MPI_COMM_WORLD will have a generation count of y after recovery, with $y > x$. A generation count is not a feature an end-user has to be aware of, but the term eases the definition of the following two message modes:

- FTMPI_MSG_MODE_RESET: This mode specifies, that a message sent from process a to process b using a communicator with a generation count x cannot be received with any communicator having the generation count y, even if the processes a and b are both surviving processes. This mode basically implies, that all ongoing and

posted messages are discarded as soon as a recovery operation has been started.

> *Rationale:* This message mode is usefull for all applications, which on error go back to the last consistent state in the application. As an example, going from iteration 432 (when the error occured) back to iteration 400 (the last checkpoint) implies that any message from iteration 432 would disturb and be misplaced.

- FTMPI_MSG_MODE_CONT: in this mode, the generation count is not used for message matching. Thus, a message sent from process a to process b before a failure occured, will be delivered after the recovery operation. All operations, which returned MPI_SUCCESS to a non failing process will be finished successfully after recovery.

> *Rationale:* This message mode is usefull for applications, which keep precisly track of the current state of each process and would like to minimize the roll-back necessary after recovery.

> *Advice to users:* If an application would like to receive a message which has been initiated before an error occured after the recovery operation, it has to reconstruct the communicators in the very same order like previously.

> *Advice to implementors:* An MPI implementation has to insure, that two sequences creating communicators in an identical manner in different generation counts will produce the same communicator/context ID's.

Blocking operations: A send operation which returned MPI_SUCCESS will deliver the data, even if a failure occurs before the data could reach the destination. If the return code of the send operation is MPI_ERR_OTHER, the operation will have to be repeated after the recovery procedure.

Non-blocking operations: if a non-blocking point-to-point operation returned MPI_SUCCESS to a process, which has not failed, than the operations will be finished successfully. If the according Wait/Test operations returns MPI_ERR_OTHER, the user will have to re-post the Wait/Test operation after recovery.

> *Advice to users:* For MPI_Waitall/Waitsome/Testome the user might have to check the error code in the status of the according operations to determine, which Wait/Test operations have to be reposted.

> *For discussion:* If a non-blocking operation to a failed process has been initiated, the request of this operation is 'invalid'. Any operation involving this request will return the error MPI_ERR_REQUEST. The same holds for persistent request operations.

When using the communicator mode FTMPI_COMM_MODE_SHRINK, the Wait/Test operation after recovery will contain the rank of the sender after recovery. Thus, a user might have posted the non-blocking receive operation to rank x, but the status after recovery will show, that the message is from rank y.

Operations using persistent requests are automatically 'corrected' to the new ranks of the according process.

> *Rationale:* For the message delivery using the communicator mode FTMPI_COMM_MODE_SHRINK, it is best to think of processes having a unique process ID. Thus, a communication always occurs between pairs of processes. The rank in MPI_COMM_WORLD (or any derived communicators) is in this case just the result of a mapping between process ID and the position of the process in the process sequence of the according communicator.

Figure 5.1 shows once again the relation ship between messages and generation counts of communicators.

5.1 Non-deterministic communication in MPI

*To discuss:*Difficulties can arise in communication patterns using the message mode FTMPI_MSG_MODE_CONT, if the application has a non-deterministic communication behaviour, e.g. through the usage of MPI_ANY_SOURCE. It is the responsibility of the application developer to avoid deadlocks in this case, since the MPI library can not recognize and cancel operations as long as it can not determine the destination/source process.

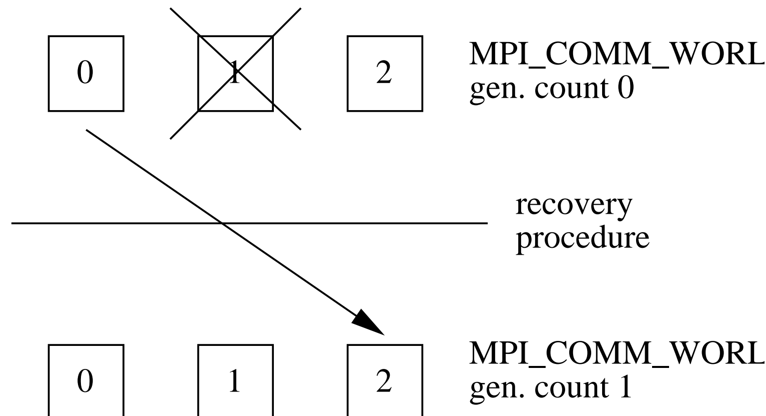

Figure 5.1: Example for a message sent and received in the same communicator however with different generation counts

Imagine for example the case, that process a posts a non-blocking receive operation from process b. Process b fails, before the data transmission can be finished. If the receive operation has been posted using a specified sender, the MPI-library can 'cancel' this operation and declare the request to be invalid. However, if the receive operation has been posted using MPI_ANY_SOURCE and no other process is sending a message which can match the posted receive, the application will deadlock.

> *Advice to users:* The usage of MPI_ANY_SOURCE should be avoided to the greates possible extent when using the message mode FTMPI_MSG_MODE_CONT.

> *For discussion:* A possibility would be to give the user an attribute after the recovery operation, which contains all request-handles which the system could not dissolve, especially the ones which have been posted using MPI_ANY_SOURCE. It is than the responsibility of the user to either cancel the request or let the communication continue.

1 ## 5.2 Collective operations

2 This section discusses the various options available for collective operations.
3 While it possible to define, when a point-to-point operation has failed or
4 succeeded, it is a lot more difficult to make a similar definition for collective
5 operations. The major question is dealing with the problem what guarantee
6 the MPI library is making to the application with respect to the fact, that

7 - everybody has the same return code for the collective operation (e.g.
8 everybody succeeds or every process returns an error)
9 - the recv data buffers are either correct on all processes or not touched
10 on any of them.

11 Therefore, FT-MPI specifies two different modes how to handle collective
12 operations:

13 1. FTMPI_COLL_MODE_ATOMIC: this mode gives strong guarantees,
14 that either every process reports an error or none.
15 2. FTMPI_COLL_MODE_NONATOMIC: no guarantee is given, that all
16 processes involved in the collective operation are returning the same
17 code. Some processes might report, that the operation has succeeded,
18 while others report an error.

> *Advice to users:* The atomic mode seems very appealing to end-users, because of the strong guarantees it is giving. Users should however be aware of, that this strong guarantee is coming at the price of higher memory consumption and higher execution time
19 for the collective operations.
20

21 Two features make the non-atomic mode still usable in fault tolerant
22 applications:

23 - First, the definition of collective operations in MPI-1 and MPI-2 is
24 such that, the input buffers are not modified in a collective operation.
25 Thus, a collective operation can easily be repeated by the application.
26 Exception: the usage of the MPI_IN_PLACE argument of MPI-2.
27 - Second, a similar behavior like the atomic mode can be achieved by
28 adding a barrier operation after a collective operation. Using this tech-
29 nique, the user has the choice to 'define' which operations he would
30 need having atomic behaviour and which not. This might have dra-
31 matic impact on the application performance.

1
2

Advice to users: It is highly recommended not to use MPI_IN_PLACE for the non-atomic collective mode.

Advice to implementors: All MPI collective operations can be implemented by allocating internally a temporary receive buffer, executing the collective operations using the temporary receive buffer, executing a two-phase commit algorithm to ensure that every process has finished successfully and just copy in case that every process succeeded the result from the temporary receive buffer into the user-provided buffer. However, there are faster algorithms for some operations available.

3
4

Chapter 6

Miscellany

6.1 New Attributes

The FT-MPI specification introduces some new attributes, reflecting the extended functionality of this specifications. These attributes can be retrieved for the default communicators MPI_COMM_WORLD and MPI_COMM_SELF, and can not be altered by the user.

- FTMPI_RECOVERY_MODE: This attribute returns the current recovery mode.
- FTMPI_COMM_MODE: This attribute returns the current communicator mode.
- FTMPI_MSG_MODE: This attribute returns the current message mode.
- FTMPI_COLL_MODE: This attribute returns the current collective communication mode.
- FTMPI_NUM_FAILED_PROCS: This attribute returns the number of failed processes since the last recovery operation.
- FTMPI_ERROR_FAILURE: This attribute returns an error code. Using MPI_Error_string and the error code provided in the attribute, the user can retrieve a string containing the ranks of the failed processes in MPI_COMM_WORLD. As with the previous attribute, the values provided by these attributes are always replaced by a new one, every time the recovery function is called.

Rationale: In might be desireable in later versions of the specification to add functionality to control the recovery, communicator, message and collective modes. Once again, the current specification tries to avoid the introduction of new functions to the largest possible extent.

6.2 New return code for MPI_Init

To give an MPI process the possibility to discover, whether it is a replacement for another process (e.g. in case of the communicator mode FTMPI_COMM_MODE_REBUILD), MPI_Init returns on these processes instead of MPI_SUCCESS the new return code MPI_INIT_RESTARTED_NODE;

Advice to users: If users want to avoid the usage of this new constant, they can retrieve the same information using a static constant and executing an allgather operation after the initialization (for the new processes) and after recovery (for the surviving processes) respectively.

6.3 Fault tolerance and error-handlers

*For discussion:*One of the major features of MPI is its ability to write libraries independently of certain applications. One of the key aspects in the specification of FT-MPI is to give library developers the possibility to write fault-tolerant libraries independent without having a specific application in mind.

MPI supports the concept of error-handlers. An application can register a function as an error-handler, which is than called in case an error occurs with the communicator, to which the error handler has been attached to. While the concept of error handlers is very convinient in MPI-1 and MPI-2, it is just partially convinient to write fault tolerant libraries. It's major drawback is, that just **one** error handler can be registered at a time.

Imagine a simple example: an application generates a subcommunicator of MPI_COMM_WORLD to perform certain operations. This subset of processes is using a library, which makes a duplicate of the subcommunicator to avoid interfering with the application messages. In case a process failes, all subcommunicators are 'destroyed', thus the user might want to write an error handler, which regenerates its subcommunicator 'automatically'

for him. The library again would like to register its own error-handler to generate the duplicate of the subcommunicator. By doing this, the library however replaces the error-handler instantiated by the application.

An improved version of error-handlers would allow to register a sequence of functions, which are called according to the order how they have been registered. Since neither MPI-1 nor MPI-2 are providing such a mechanism, the current FT-MPI specification suggests the following model.

As the last step of the recovery procedure, the MPI library will call all attribute delete functions attached to MPI_COMM_WORLD. Since the deletion of this communicator is erroneous, there is no danger that these functions are accidentally called when freeing the communicator.

Rationale: A similar mechanism is provided in MPI-2 to allow user defined functions to be executed in MPI_Finalize (similarly to the UNIX *atexit*() command). In contrary to this specification MPI-2 uses however MPI_COMM_SELF. The attribute copy functions can not be used for the discussed purpose, since duplicating MPI_COMM_WORLD is allowed.

Advice to implementors: Although not clearly specified in MPI-1 and MPI-2, the sequence of calling the attribute delete functions should match the order of how they have been registered.

1 Chapter 7

2 Conclusions

Bibliography

[1] MPI Forum: *MPI: A Message-Passing Interface Standard.* Document for a Standard Message-Passing Interface, University of Tennessee, 1995.

[2] MPI Forum: *MPI2: Extentions to the Message-Passing Interface Standard.* Document for a Standard Message-Passing Interface, University of Tennessee, 1997.