

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

Graham E. Fagg, Edgar Gabriel, George Bosilca,  
Thara Angskun, Zizhong 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, angsun, zchen, pjesa, london, dongarra}@cs.utk.edu

## 1. Motivation

### 1.1 Trends in High Performance Computing

Today, end-users and application developers of high performance computing systems have access to larger machines and more processors than ever before. 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 in the next few years.

A critical issue for systems consisting of such large numbers of processors is the ability of the machine to deal with process failures. Based on current experience with high-end systems, it can be concluded, that a 100,000-processor machine will experience a processor 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 far 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 both the hardware and the system software layers.

### 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.

However, the current MPI specification does not deal with the case where one or more process failures occur during runtime. 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 is mainly the to allow 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. MPI implementations focusing on different aspects of parallel and distributed computing are becoming readily available. The manufacturers of high-performance computing systems are enhancing the performance of MPI-1 functions as well as improving support for 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 topology hierarchies, security aspects as well as user-friendliness. Examples are PACX-MPI[3], MPICH-G2[14], Stampi[19], MetaMPICH[18] or GridMPI[8].

Additional projects that deal with various aspects of fault-tolerance include; MPI/FT[15], MPI-FT[17], MPICH-V[5] and 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, there is a discrepancy between the capabilities of current high performance computing systems and the most widely used parallel programming paradigm. While the machines are improving their robustness (hardware, network, operating and file systems) the MPI specification does not leave room for fully exploiting the capabilities of the current architectures. When considering machines with tens of thousand of processors, the only currently available fault tolerance handling technique, checkpoint/restart, has its performance and conceptual limitations. In fact, one of the main reasons for many research groups to prefer the PVM[4] communication library to MPI is its capability to handle process failures.

If today's and tomorrows high performance computing resources were to be used as a means to perform single, large scale simulations and not solely as a platform for high throughput computing, extending the core communication library of HPC systems to deal with aspects of fault-tolerance becomes 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 only assumption FT-MPI makes about the first two steps is that the run-time environment discovers failures and all remaining processes in the parallel job are notified about these 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: 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 FT-MPI specification tackles 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*, and the third by the *message mode* respectively the *collective communication mode*.

The recovery mode defines how the recovery procedure can be started. Currently, three options are 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 be compacted so that they remain contiguous (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 the status of messages when errors occur. In the first mode, all messages in transit are cancelled by the system. This mode is mainly useful for applications, which on error rollback 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 the rollback. The second mode completes the transfer of all messages after the recovery operation,

with the exception of the messages to and from the failed processes. This mode requires that the application keep detailed information of the state of each process, minimizing the rollback procedure. Similar modes are available for collective operations, which can either be executed in an atomic or a non-atomic fashion.

### 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 HARNESST[2][7] framework. HARNESST (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 more than a theoretical framework.

The currently available functionality includes the full MPI-1.2 specification, as well as several sections of the MPI-2 document. Furthermore, many efforts have been invested in optimizing the collective operations and the derived datatype section of MPI. A multi-protocol device supporting TCP/IP and various other protocols (e.g. shared memory, Myrinet) is currently in the testing phase. The library has been tested on a wide variety of platforms, operating systems and compilers, e.g. IA-32 LINUX and Windows platforms, AMD 64-bit architectures, SUN, SGI and HP platforms.

In various benchmarks and application scenarios, UTK's implementation has proven that the performance of FT-MPI is comparable to the current state-of-the art MPI libraries during a fault-free execution. This indicates, that as long as no error occurs, the new specification does not harm the performance of applications. Furthermore, the results 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 abstracts the features needed to handle fault-tolerance 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 scenarios

Simultaneously to the development of the specification and UTK's 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 such as the Reed-Solomon Algorithm on other processes. When an error occurs, the application need not be restarted, since the additional information can be used to reconstruct the data of the failed processes. This technique improves the performance of the application for large number of processes, 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,
- Cholesky factorization.

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

A fault-tolerant manager-worker framework, which does not use in-memory checkpointing, has been developed [6]. The key point of this framework is to show, that all applications, which can be written using a manager worker paradigm (e.g. parameter 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).

Even with checkpoint/restart techniques based on file operations, the FT-MPI approach still offers advantages to the end-users, as shown for the parallel spectral transform shallow water code (PSTSWM)[23]. In case an error occurs, the application could re-spawn the failed processes and reconstruct the missing data based on the last checkpoint files. The advantage compared to conventional checkpointing techniques is, that the application does not have to be stopped and restarted as an entity. Thus, the application will still finish correctly despite of an error without any user interaction.

Recent work by Geist and Engelman[1] presented new algorithms for solving partial differential equations, using a property referred too as 'natural fault tolerance'. Based on mesh-less methods and chaotic relaxation, Geist and Engelman show, that the algorithm still converges correctly, despite the loss of a marginal number of processes. In their approach, the failing processes do not have to be replaced, e.g. using the BLANK mode of the FT-MPI specification. 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 various approaches to handling failures, while still allowing users flexibility to handle fault-tolerance according to the requirements of their applications.

## 5. Summary

In this paper we have presented 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 usage scenarios.

The current specification is in the spirit of the MPI 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 introduce any constraints on application developer for handling fault tolerance. Instead, FT-MPI offers a rich set of techniques to handle processes failures and defines the status of MPI objects if a failure occurs. This allows application developers to take the specific characteristics of their application into account and use the best-suited method 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] AI 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. Papadopoulos, S. Scott, V. Sunderam, "HARNES: 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, HARNES 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 Jack J. Dongarra, Domenico Laforenza, Salvatore Orlando (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 Forum: '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 1<sup>st</sup> 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.ifbs.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, MagPle: 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.
- [23] P. H. Worley and B. Toonen: 'A User's Guide to PSTSWM', July 1995.

FT-MPI: Proposal for Extensions to the Message-Passing  
Interface for Process Fault-Tolerance

May 17, 2004

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

## **Abstract**

This document describes extensions to the MPI-1.2 and MPI-2 standards for introducing process fault-tolerance in MPI. This includes some new semantics for communicators as well as several new attributes.

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Background . . . . .	2
1.2	Organization of this Document . . . . .	3
<b>2</b>	<b>Basic fault-tolerance issues</b>	<b>4</b>
<b>3</b>	<b>Recovery Modes</b>	<b>7</b>
3.1	Pathological failures . . . . .	8
<b>4</b>	<b>Communicator modes</b>	<b>10</b>
4.1	FTMPI_COMM_MODE_BLANK . . . . .	12
4.2	FTMPI_COMM_MODE_SHRINK . . . . .	14
<b>5</b>	<b>Message modes</b>	<b>15</b>
5.1	Non-deterministic communication in MPI . . . . .	17
5.2	Collective operations . . . . .	19
<b>6</b>	<b>Miscellany</b>	<b>21</b>
6.1	New Attributes . . . . .	21
6.2	New return code for MPI_Init . . . . .	22
6.3	Fault tolerance and error-handlers . . . . .	22

# 1 Chapter 1

## 2 Introduction

### 3 1.1 Background

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

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

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

1 which is to immediately abort the application. The second possibility is to  
2 hand control back to the user application (if possible) without guarantee-  
3 ing, that any further communication can occur. The latter mode mainly  
4 has the purpose of giving the application the possibility to close all files  
5 correctly, and maybe write a per-process checkpoint file etc., before exiting  
6 the application.

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

## 11 **1.2 Organization of this Document**

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

## 1 Chapter 2

# 2 Basic fault-tolerance issues

3 Fault tolerance usually covers three steps:

- 4 • Fault detection
- 5 • Notification
- 6 • Recovery

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

13 Notification deals with the problem of how the other MPI processes of a  
14 parallel job get informed about the failure event. FT-MPI makes no assump-  
15 tions **when** the processes are notified nor does it assume, that all processes  
16 are notified simultaneously. FT-MPI only specifies, that all processes of a  
17 parallel job do receive a notification about death events.

18 The notification of failed processes are passed to the MPI application  
19 through a special error code. For achieving the largest possible conformance  
20 to the MPI-1 and MPI-2 specification, FT-MPI does not introduce any new  
21 error codes, but defines, that `MPI_ERR_OTHER` is only to be used to signal  
22 the MPI application that some process(es) have unexpectedly left the run-  
23 time environment.

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

1 state, the process is only allowed to execute certain actions. These actions  
2 depend on various other parameters and are detailed later in the document.

3 *Rationale:* While the introduction of a new error-code indicating  
4 failed processes would have been desirable, currently we consider  
the requirement to have an FT-MPI specification, which allows  
execution of an application written according to the FT-MPI spec-  
ification on any regular non FT-MPI conformant implementation  
of MPI as more important than a 'cleaner' solution currently. It is  
however still desirable to introduce a separate error-code in future  
specifications.

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

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

12 The recovery procedure is considered to have two steps:

- 13 1. Recovering the MPI run-time environment and the MPI library. This  
14 step will be discussed in great detail in the following sections.
- 15 2. Recovering the application and application data: this step is consid-  
16 ered to be the responsibility of the application and **not** of the MPI  
17 library. The FT-MPI specification makes no assumptions or state-  
18 ments about how an application recovers data from one or several lost  
19 processes.

20 *Rationale:* in contrary to many currently available projects, FT-  
21 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, it is however not considered in the current  
version.

22 The FT-MPI specification addresses the following questions related to  
23 the recovery operation:

- 1     1. What are the required steps and/or options to start the recovery proce-  
2         dure once the processes are in the 'FAILURE RECOGNIZED' status?
- 3     2. What is the status of MPI objects and processes after recovery?
- 4     3. What is the status of ongoing communication (point-to-point com-  
5         munication as well as collective operations) after recovering from a  
6         failure?

7         The first question is handled by the **recovery mode** (FTMPI\_RECOVERY-  
8         \_MODE), the second by the **communicator mode**(FTMPI\_COMM\_MODE)  
9         and the third by the **message mode**(FTMPI\_MSG\_MODE) and respec-  
10        tively the **collective communication mode** (FTMPI\_COLL\_MODE).

# 1 Chapter 3

## 2 Recovery Modes

3 The user has three possibilities on how the recovery procedure can be initi-  
4 ated:

- 5 1. FTMPI.RECOVERY\_MODE\_AUTO: as soon as the MPI library re-  
6 alizes, that a death event has occurred, it automatically starts the  
7 recovery process. No interaction from the application is required. Af-  
8 ter the recovery has been successfully finished, the error handler of  
9 MPI\_COMM\_WORLD is called, since other communicators are not  
10 available after recovery. The state of communicators, groups and other  
11 objects are defined in later sections.
- 12 2. FTMPI.RECOVERY\_MODE\_MANUAL: as with any other error, the  
13 MPI library calls the error handler attached to the current commu-  
14 nicator. The user is however not allowed to call any MPI function  
15 involving communication before the recovery has been started.

16 To start the recovery, the user has to call MPI\_Comm\_dup on MPI-  
17 \_COMM\_WORLD. The input argument of MPI\_Comm\_dup should be  
18 MPI\_COMM\_WORLD, the output argument is undefined and should  
19 be ignored by the application.

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

22

*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 versions to avoid the dual functionality of MPI\_Comm\_dup is highly recommended.

1  
2

- 3 3. FTMPIRECOVERY\_MODE\_IGNORE: in this mode, the recovery  
4 procedure does not have to be initiated ever. This requires that no  
5 communication with the dead processes is attempted once the failure  
6 is detected and the MPI application notified. Communication involv-  
7 ing dead processes (point-to-point operations, collective operations as  
8 well as communicator creations) will raise an error and will not be  
9 executed.

*Rationale:* This mode has been designed to handle two separate issues. First, since the recovery procedure is a collective operation, it is desirable to avoid this collective operation for large numbers of processors (e.g. 100,000). Second, there is a class of applications often referred to as 'naturally fault-tolerant' which do not require any special functionality such as checkpointing at the application level to deal with failed processes.

10  
11

## 12 3.1 Pathological failures

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

- 15 • All processes of an MPI job have failed before a recovery operation  
16 could be started.
- 17 • The MPI library can not locate and access additional resources on  
18 which to re-spawn failed processes.
- 19 • The Runtime system has suffered a pathological failure itself and can-  
20 not re-spawn new processes.

*Advice to implementors:* The Runtime system should be designed in such a way that it is itself fault tolerant and reliable across a wide range of possible conditions. A minimum requirement would be a runtime system failure should not effect an already running fault free application, even if the failure prevents new processes/applications from being spawned/started.

<sup>1</sup>  
<sub>2</sub>

# 1 Chapter 4

## 2 Communicator modes

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

- 8 • Datatypes (MPI\_Datatype)
- 9 • Operations (MPI\_Op)
- 10 • Error handlers (MPI\_Errhandler)
- 11 • Info objects (MPI\_Info, MPI-2)

12 The following list shows the objects which are destroyed during the re-  
13 covery procedure:

- 14 • Groups (MPI\_Group)
- 15 • Communicators (MPI\_Comm)
- 16 • Windows (MPI.Win, MPI-2)
- 17 • Files (MPI\_File, MPI-2)

18 Requests are in this context 'special' object, their behaviour is dependent  
19 on the message mode, this is further explained in section 5. Windows and  
20 Files will be handled in more details in future versions of the specification.

*Rationale:* Although groups are considered to be local objects in MPI, they usually contain 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.

1  
2  
3  
4  
5

After the recovery operation, the user has access to the same non-local operations and state as was available after the initial MPI\_Init call. These are:

6  
7

- 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.

8  
9  
10  
11  
12  
13

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:

14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26

1. FTMPI\_COMM\_MODE\_ABORT: as in MPI-1 and MPI-2. The MPI library will abort the execution gracefully 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 **which resources** the new processes are allocated to. (This would be Runtime system dependent).
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.

- 1 4. FTMPI\_COMM\_MODE\_SHRINK: failed processes will not be replaced.  
 2 The size of MPI\_COMM\_WORLD will be adjusted to the number of  
 3 surviving processes. This might also alter the ranks of some processes  
 4 in MPI\_COMM\_WORLD. FT-MPI requires that the sequence of surviving  
 5 processes is identical before and after recovery. Figure 4.1 shows  
 6 an example where two out of four processes fail, and the ranks of the  
 7 remaining processes after the recovery.

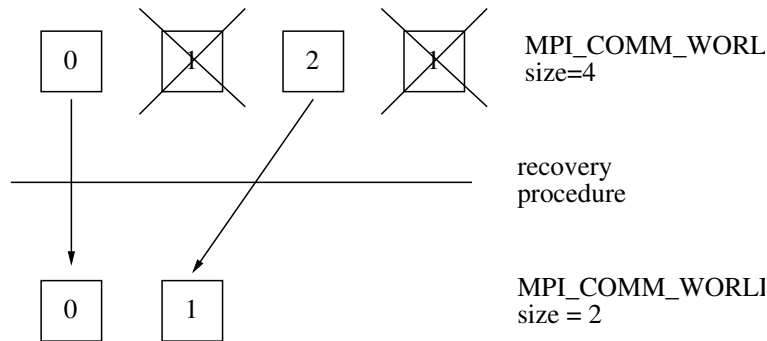


Figure 4.1: Example of the communicator mode FTMPI\_COMM\_MODE\_SHRINK.

8 FTMPI\_COMM\_MODE\_ABORT and FTMPI\_COMM\_MODE\_REBUILD  
 9 require no changes to the MPI-1 and MPI-2 specification after recovery.  
 10 The communicator modes FTMPI\_COMM\_MODE\_BLANK and FTMPI-  
 11 \_COMM\_MODE\_SHRINK introduce some new aspects to MPI and are de-  
 12 scribed more fully below.

#### 13 4.1 FTMPI\_COMM\_MODE\_BLANK

14 **Point-to-point operations** A blank process is defined to behave like  
 15 MPI\_PROC\_NULL in the MPI-1 specification. I.e., that sending a message  
 16 to a blank process will not raise an error, however no data is transmitted.  
 17 Receive operations from a blank processes will return a null-status (see sec-  
 18 tion 3.11 in MPI-1), the receiver buffer is unchanged.

19 **Collective operations** For collective operations, two different issues have  
 20 to be taken into account for the blank mode. If the root of one of the rooted  
 21 collective operations (MPI\_Bcast, MPI\_Reduce, MPI\_Gather(v), MPI\_Scatter(v))

1 is a blank process all processes will return immediately. No input or output  
2 buffer is modified.

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

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

16 The rules for non-rooted operations can be directly derived from the  
17 rules for rooted operations. The implementation has to ensure, that for  
18 operations, which are implemented as a combination of other collective op-  
19 erations (e.g. MPI\_Allreduce implemented as an MPI\_Reduce followed by an  
20 MPI\_Bcast) a temporary root node is chosen, which is not a blank process.

21 **Group and Communicator creation functions** All operations defined  
22 in MPI-1 and MPI-2 for deriving new groups and communicators are valid  
23 for blank processes as well. This includes:

- 24 • it is valid to split communicators containing blank processes, assuming  
25 that they are providing the color MPI\_UNDEFINED.
- 26 • it is valid to derive a group from a communicator which contains blank  
27 processes
- 28 • it is valid to include/exclude blank processes from a group
- 29 • it is valid to generate new communicators from groups containing or  
30 excluding blank processes.

31 The group and communicator comparison functions MPI\_Group\_compare  
32 and MPI\_Comm\_compare need not be able to distinguish between two blank  
33 processes.

1 All topology functions might include blank processes. The outcome of  
2 the topology functions returning the ranks of neighbor processes might be  
3 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. How-  
ever, this would be equal to ignoring the 'distance' argument pro-  
vided by the user, or at least a redefinition of it.

## 6 **4.2 FTMPI.COMM.MODE.SHRINK**

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

# 1 Chapter 5

## 2 Message modes

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

- 8 1. All messages from and to dead processes are discarded, independent  
9 of recovery, communicator or message mode.
- 10 2. All collective operations will stop immediately and all messages ini-  
11 tiated by collective operations will be discarded, independent of the  
12 recovery, communicator or message mode. In the following subsection,  
13 we will furthermore discuss the behavior of collective operations while  
14 an error occurs.

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

- 22 • `FTMPI_MSG_MODE_RESET`: This mode specifies, that a message  
23 sent from process  $a$  to process  $b$  using a communicator with a gen-  
24 eration count  $x$  cannot be received with any communicator having  
25 the generation count  $y$ , even if the processes  $a$  and  $b$  are both sur-  
26 viving processes. This mode basically implies, that all ongoing and

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

3 *Rationale:* This message mode is useful for all applications, which  
4 on error go back to the last consistent state in the application. As  
5 an example, going from iteration 432 (when the error occurred)  
6 back to iteration 400 (the last checkpoint) implies that any mes-  
7 sage from iteration 432 would disturb and be misplaced.

- 8 • FTMPI\_MSG\_MODE\_CONT: in this mode, the generation count is  
9 not used for message matching. Thus, a message sent from process  
10 *a* to process *b* before a failure occurred, will be delivered after the  
11 recovery operation. All operations, which returned MPI\_SUCCESS to  
12 a non failing process will be finished successfully after recovery.

10 *Rationale:* This message mode is useful for applications, which  
11 keep precisely track of the current state of each process and would  
12 like to minimize the roll-back necessary after recovery.

12 *Advice to users:* If an application would like to receive a message  
13 which has been initiated before an error occurred after the recovery  
14 operation, it has to reconstruct the communicators in the identical  
15 order as used previously.

14 *Advice to implementors:* An MPI implementation has to insure,  
15 that two sequences creating communicators in an identical manner  
16 in different generation counts will produce the same communi-  
17 cator/context ID's.

16 **Blocking operations:** A send operation which returned MPI\_SUCCESS  
17 will deliver the data, even if a failure occurs before the data could  
18 reach the destination. If the return code of the send operation is  
19 MPI\_ERR\_OTHER, the operation will have to be repeated after the  
20 recovery procedure.

21 **Non-blocking operations:** if a non-blocking point-to-point operation  
22 returned MPI\_SUCCESS to a process, which has not failed, then the  
23 operations will be finished successfully. If the according Wait/Test  
24 operations returns MPI\_ERR\_OTHER, the user will have to re-post  
25 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.

1  
2

*For discussion:* If a non-blocking operation to a failed process has been initiated, the request of this operation is 'invalid' and is internally canceled/freed. Any operation involving this request will return the error MPIERR\_REQUEST. The same holds for persistent request operations.

3  
4

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  $y$ .

5  
6  
7  
8  
9  
10  
11

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.

12  
13

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

14  
15

## 16 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.

17  
18  
19  
20  
21  
22

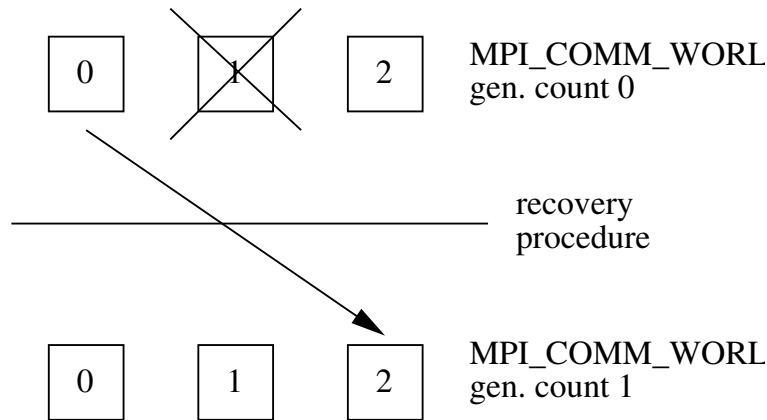


Figure 5.1: Example for a message sent and received in the same communicator however with different generation counts

1        Imagine for example the case, that process *a* posts a non-blocking receive  
 2        operation from process *b*. Process *b* fails, before the data transmission can be  
 3        finished. If the receive operation has been posted using a specified sender,  
 4        the MPI-library can 'cancel' this operation and declare the request to be  
 5        invalid. However, if the receive operation has been posted using MPI\_ANY-  
 6        \_SOURCE and no other process is sending a message which can match the  
 7        posted receive, the application would deadlock.

8        *Advice to users:* The usage of MPI\_ANY\_SOURCE should  
 9        be avoided where ever possible when using message mode  
 FTMPI\_MSG\_MODE\_CONT.

10        *For discussion:* A possibility would be to give the user an attribute  
 11        after the recovery operation, which contains all request-handles  
 which the system could not resolve, especially the ones which have  
 been posted using MPI\_ANY\_SOURCE. It is 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 is possible to define when a point-to-point operation has failed or  
4 succeeded, it is much more difficult to make a similar definition for collective  
5 operations. The major question is dealing with what guarantee the MPI  
6 library is making to the application with regards to the following issues:

- 7 • unanimous results. Everybody has the same return code for the col-  
8 lective operation (e.g. every process succeeds or every process returns  
9 an error, but not a mix of results)
- 10 • consistence of users memory buffers. The recv data buffers are either  
11 correct on all processes or not touched on any of them.

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

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

20 *Advice to users:* The atomic mode may seem very appealing to  
21 end-users, due to the strong guarantees it offers. Users should  
22 however be aware that this strong guarantee comes at the price  
23 of higher memory consumption and higher execution time for the  
24 collective operations.

25 Two features make the non-atomic mode still usable in fault tolerant  
26 applications:

- 27 • First, the definition of collective operations in MPI-1 and MPI-2 is  
28 such that, the input buffers are not modified in a collective operation.  
29 Thus, a collective operation can easily be repeated by the application.  
30 Exception: the usage of the MPI\_IN\_PLACE argument of MPI-2.
- Second, a similar behavior to the atomic mode can be achieved by  
adding a barrier operation after a collective operation. Using this tech-  
nique, the user has the choice to 'define' which operations he would

1        need to have atomic behaviour and which not. This might have dra-  
2        matic impact on the application performance.

3                *Advice to users:* It is highly recommended not to use  
4                MPI\_IN\_PLACE for the non-atomic collective mode.

5                *Advice to implementors:* All MPI collective operations can be  
6                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 then copy from the  
              temporary receive buffer into the user-provided buffer. However,  
              there are faster algorithms for some operations available.

# 1 Chapter 6

## 2 Miscellany

### 3 6.1 New Attributes

4 The FT-MPI specification introduces some new attributes, reflecting the ex-  
5 tended functionality of this specifications. These attributes can be retrieved  
6 for the default communicators `MPI_COMM_WORLD` and `MPI_COMM_SELF`,  
7 and can not be altered by the user.

- 8 • `FTMPI_RECOVERY_MODE`: This attribute returns the current re-  
9 recovery mode.
- 10 • `FTMPI_COMM_MODE`: This attribute returns the current communi-  
11 cator mode.
- 12 • `FTMPI_MSG_MODE`: This attribute returns the current message mode.
- 13 • `FTMPI_COLL_MODE`: This attribute returns the current collective  
14 communication mode.
- 15 • `FTMPI_NUM_FAILED_PROCS`: This attribute returns the number of  
16 failed processes since the last recovery operation.
- 17 • `FTMPI_ERROR_FAILURE`: This attribute returns an error code. Us-  
18 ing `MPI_Error_string` and the error code provided in the attribute, the  
19 user can retrieve a string containing the ranks of the failed processes  
20 in `MPI_COMM_WORLD`. As with the previous attribute, the values  
21 provided by these attributes are always replaced by a new one, every  
22 time the recovery function is called.

*Rationale:* It might be desirable 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.

1  
2

## 3 6.2 New return code for MPI\_Init

4 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-  
5 \_COMM\_MODE\_REBUILD), MPI\_Init returns on these processes instead of  
6 MPI\_SUCCESS the new return code MPI\_INIT\_RESTARTED\_NODE;  
7

*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.

8  
9

## 10 6.3 Fault tolerance and error-handlers

11 *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  
12 specification of FT-MPI is to give library developers the possibility to write  
13 fault-tolerant libraries independently without having a specific application  
14 in mind.  
15

16 MPI supports the concept of error-handlers. An application can register  
17 a function as an error-handler, which is then called in case an error occurs  
18 with the communicator, to which the error handler has been attached to.  
19 While the concept of error handlers is very convenient in MPI-1 and MPI-2,  
20 it is just partially convenient to write fault tolerant libraries. Its major  
21 drawback is, that just **one** error handler can be registered at a time.

22 Imagine a simple example: an application generates a sub-communicator  
23 of MPI\_COMM\_WORLD to perform certain operations. This subset of processes  
24 is using a library, which makes a duplicate of the sub-communicator  
25 to avoid interfering with the application messages. In case a process fails,  
26 all sub-communicators are 'destroyed', thus the user might want to write  
27 an error handler, which regenerates its sub-communicator 'automatically'

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

4 An improved version of error-handlers would allow an application to  
5 register a sequence of functions, which are called according to the order that  
6 they were registered initially. Since neither MPI-1 nor MPI-2 are providing  
7 such a mechanism, the current FT-MPI specification suggests the following  
8 model.

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

13 *Rationale:* A similar mechanism is provided in MPI-2 to allow  
14 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 func-  
tions can not be used for the discussed purpose, since duplicating  
MPI\_COMM\_WORLD is allowed.

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

# 1 Bibliography

- 2 [1] MPI Forum: *MPI: A Message-Passing Interface Standard*. Document  
3 for a Standard Message-Passing Interface, University of Tennessee,  
4 1995.
- 5 [2] MPI Forum: *MPI2: Extentions to the Message-Passing Interface Stan-*  
6 *dard*. Document for a Standard Message-Passing Interface, University  
7 of Tennessee, 1997.