# Fault Tolerance in MPI Programs*

William Gropp and Ewing Lusk

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{gropp,lusk}@mcs.anl.gov

**Abstract.** This paper examines the topic of writing fault-tolerant MPI applications. We discuss the meaning of fault tolerance in general and what the MPI Standard has to say about it. We survey several approaches to this problem, namely checkpointing, restructuring a class of standard MPI programs, modifying MPI semantics, and extending the MPI specification. We conclude that within certain constraints, MPI can provide a useful context for writing application programs that exhibit significant degrees of fault tolerance.

## 1 Introduction

As modern supercomputers scale to hundreds or even thousands of individual nodes, the Message Passing Interface (MPI) remains a straightforward and effective way to program them. At the same time the larger number of individual hardware components means that hardware faults are more likely to occur during long-running jobs. Users naturally want their programs to adapt to hardware faults and continue running. This ideal is clearly unattainable in general (e.g. if *all* nodes fail) but users still can achieve a significant degree of fault tolerance for their MPI programs. This paper explores several approaches that are possible within the context of MPI.

In Section 2, we briefly survey some systems that provide fault tolerance. In Section 3, we clarify the relationship between fault tolerance and the MPI Standard, MPI implementations, and parallel algorithms. We claim that fault tolerance is a property of a program, not of an API specification or an implementation. In Section 4, we detail what the MPI Standard says that is related to fault tolerance issues. In Section 5, we describe several approaches to achieving fault tolerance in MPI programs, namely, checkpointing, using MPI intercommunicators to write a class of fault-tolerant MPI programs, modifying the semantics of existing MPI functions to provide more fault-tolerant behavior, and defining extensions to the MPI specification that support the writing of fault-tolerant MPI programs. We summarize our conclusions in Section 6.

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

## 2 Existing Systems

Researchers have explored a number of different approaches to providing fault tolerance in MPI programs. One of the most recent, MPICH-V [2], is also one of the most complete. It provides complete checkpointing and message logging to enable replacement of aborted processes; the checkpoints avoid reconstructing computations from the beginning through the message logs. It also provides scalability but requires a reliable subsystem for the checkpoints and message logs, as well as for the "dispatcher" process. A similar approach is taken in the LAM-based MPI-FT [11]. MPICH-V demonstrates the cost (approximately doubling of communication times) of providing full recovery in all situations.

Egida [14] is another MPICH-based system using logs for transparent recovery. It incorporates its own language to allow the user to express various responses to faults. A complete but less scalable approach is taken in MPI/FT (TM) [1].

FT-MPI [4, 5] explores the approach of modifying some of the standard MPI semantics. We discuss this approach further in Section 5.3.

For a discussion of some algorithmic approaches to fault tolerance, see [6] in this collection.

A large body of knowledge known as *transaction processing* is concerned with fault tolerance and the various solutions to the problem of reliable computing in the presence of failures (see, e.g., [7]). Methods for fault-tolerant transaction processing often rely on maintaining redundant or duplicated state between two parties (often processes); if one of these processes fails, the other can continue the computation because no information has been lost.

Most approaches to fault tolerance have a similar set of requirements:

- Failure can be detected
- Information (state) needed to continue the computation is available
- The computation can be restarted.

In this paper, we discuss each of these requirements in the context of MPI.

## 3 What Does "Fault Tolerance" Mean?

Misunderstanding about the relationship between MPI and fault tolerance is evident in assertions like "MPI is not fault tolerant." This statement is not actually well formed and so is neither true nor false. Let us examine what motivates such assertions and what can be stated with more precision.

A common misconception about MPI is that the MPI Standard itself mandates that if any MPI process dies, then all the MPI processes in the job must die as well. This is not true. The basis for this misconception is easily understandable. The standard says (in Section 7.2 of [13] or Section 7.5.1 of [15]) that the default error handler on the communicator `MPI_COMM_WORLD` is the built-in one called `MPI_ERRORS_ARE_FATAL`. (See Section 4.2 for a discussion of error handlers attached to communicators.) Thus, if one takes no particular action with respect to error handling, when a process exits before calling `MPI_Finalize`, the others are indeed required to detect this condition and exit as well. The MPI Forum decided that this would probably be the most useful default behavior, particularly for new users. (And when the MPI Forum was deliberating, *all* users were new.)

Fault tolerance is a property; what is it a property *of*? It is not a property of MPI itself, since MPI is a specification of an API (application programmer interface). The MPI Standard describes how to write a correct parallel program. It tends, for the most part, to assume that this program will execute on reliable hardware. Considerable latitude is granted the implementation on how hardware faults will be handled; we discuss this in Section 4.

Is fault tolerance thus a property of an MPI implementation? No, since no implementation can ensure that any program is immune from all faults.

We claim that fault tolerance is a property of an MPI *program* coupled with an MPI *implementation*. For simplicity we include as part of the MPI implementation the hardware and software environment the program is running in. Thus, a particular MPI implementation (no more or less standard-conforming than any other) may still extend (or restrict) the class of MPI programs that will exhibit various degrees of fault tolerance when linked with a particular MPI implementation and executed.

By fault tolerance, most people mean that a fault-tolerant program, linked to a matching implementation, can "survive" (in some sense we discuss shortly) a failure of the hardware infrastructure, such as a network failure or a machine crash. As we mentioned, this ideal is not completely attainable in general, but many useful partial achievements in this direction are attainable.

"Survival" is a broad term encompassing various levels of action. The highest level of survival is that the MPI implementation automatically recovers from some set of faults and the MPI program, regardless of its structure, continues without significant change to its behavior. A second level of survival is that the program is notified of the problem and is prepared to take corrective action. (This approach is described in Section 5.2.) A third level of survival is that certain MPI operations, although not all, become invalid. (This approach is described in Sections 5.3 and 5.4.) In each of these three cases, the program arranges for the nonfailing processes to retain enough of the program state held by the failed process for the overall computation to proceed. A fourth level of survival is that a program can abort and be restarted from a checkpoint (Section 5.1). Here the states of all processes are saved outside the processes themselves, typically on disk. Finally, combinations of these approaches may be used.

## 4 The MPI Standard and Fault Tolerance

What does the MPI Standard say that is relevant to fault tolerance? One might think that it says nothing, since as an API specification it describes the behavior of correct MPI programs running on reliable hardware; it is not a specification for a complete execution environment and does not specify the behavior of incorrect programs. Nonetheless, the MPI Standard does make a number of precise statements in this area and also provides considerable flexibility in the handling of errors. Both of these are important for understanding how to write fault-tolerant MPI programs.

### 4.1 Reliable Communication

The MPI Standard specifies reliable communication (see Section 1.1 of [12]). That is, if an MPI implementation allows a message to be delivered in a corrupted state (i.e., the contents of the received message are not identical to the contents of the message sent), then it is a nonconforming implementation. Thus, the MPI implementation is responsible for detecting and handling network faults. "Handling" means either recovering from the error through retransmission of the message or else informing the application that an error has occurred, allowing the application to take its own corrective action. Under no circumstances should an MPI application or library need to verify integrity of data received.

### 4.2 Error Handlers

Section 7.5 of [15] describes the association of error handlers to communicators. Error handlers can be either built in or user defined. The built-in error handlers are `MPI_ERRORS_ARE_FATAL`, which specifies that if an MPI function returns unsuccessfully then all the processes in the communicator will abort, and `MPI_ERRORS_RETURN`, which specifies that MPI functions will attempt (whether this attempt is successful may be implementation dependent) to return an error code (a return code not equal to `MPI_SUCCESS`). In C++, `MPI::ERRORS_THROW_EXCEPTIONS` is also defined. [1] The MPI Standard does state that `MPI_ERRORS_ARE_FATAL` is the default error handler on `MPI_COMM_WORLD` and that new communicators inherit the error handler of the parent communicator, thus leading to the common misconception that the standard requires this "all fall down" behavior.

Error handlers are set on communicators with `MPI_Comm_set_errhandler`. Not only can the default be changed from `MPI_ERRORS_ARE_FATAL` to `MPI_ERRORS_RETURN`, but users can define their own error handlers and attach them to communicators. This ability to define one's own application-specific error handler is important for the approach to fault tolerance we describe in Section 5.2. The ability to attach error handlers on a communicator basis is important for MPI's modularity: a library may want to operate with its own communicator (see [8]) and deal with errors itself in order to present them to the user in a library-specific way, while the user application program still uses `MPI_ERRORS_ARE_FATAL`.

[1] In C and C++, return codes are the return values of the MPI functions. In versions of Fortran, return codes are the value of the `ierr` parameter passed to the Fortran subroutines.

### 4.3 Errors

When MPI_ERRORS_RETURN is the active error handler, a set of errors is predefined, and implementations are allowed to extend this set. This feature is useful if an implementation wants to define certain types of error specific to fault tolerance. Indeed, the standard allows implementations considerable latitude in the handling of errors. It may not be possible for an implementation to recover from some types of errors even enough to return an error code, and such an implementation is conforming. Some conforming implementations may return errors in situations where other conforming implementations abort. If an error is returned, the standard does not require that subsequent operations succeed, or that they fail. Thus the standard allows implementations to take various approaches to the fault tolerance issue and to trade performance against fault tolerance to meet user needs. Different levels of tradeoff are exemplified in the specific implementations we described in Section 2, as well as implied in the approaches that we describe in more general terms in Section 5.

## 5 Approaches to Fault Tolerance in MPI Programs

In this section we describe several approaches to *writing* fault-tolerant programs, to be used with MPI implementations that choose not to provide "transparent" fault tolerance because of the performance cost. We cover checkpointing, using intercommunicators to restructure certain algorithms, modifying MPI semantics, and extending MPI.

### 5.1 Checkpointing

Checkpointing is a common technique that periodically saves the state of a computation, allowing the computation to be restarted from that point in the event of a failure. Checkpointing is easy to implement but is often considered expensive. After all, saving a checkpoint takes time that could otherwise be devoted to additional computation. The cost of checkpointing need not be large, however.

To estimate the cost (in running time of an application), we define the following:

$$\begin{aligned} k_0 &= \text{Cost to create and write checkpoint} \\ k_1 &= \text{Cost to read and restore checkpoint} \\ \alpha &= \text{Probability of failure} \\ t_0 &= \text{Time between checkpoints} \\ T &= \text{Total time to run, without checkpoints} \end{aligned}$$

We assume that the probability of failure is independent of time and has an exponential probability density function. For small $\alpha$, the expected running time between checkpoints, assuming at most one failure between checkpoints, can thus be approximated by

$$\begin{aligned}
E &= (1 - \alpha t_0)(k_0 + t_0) + \\
&\qquad \alpha t_0 \left( k_0 + t_0 + k_1 + \frac{1}{2} t_0 \right) \\
E &= k_0 + t_0 + \alpha \left( k_1 t_0 + \frac{1}{2} t_0^2 \right).
\end{aligned}$$

Thus, the total run time (to time $T$) is

$$E_T = \frac{T}{t_0} \left( k_0 + t_0 + \alpha \left( k_1 t_0 + \frac{1}{2} t_0^2 \right) \right).$$

We can find the optimal time between checkpoints by differentiating with respect to $t_0$ and setting the result to zero. This leads to

$$\begin{aligned}
\frac{dE_T}{dt_0} &= T \left( -\frac{k_0}{t_0^2} + \frac{1}{2} \alpha \right) \\
0 &= -\frac{k_0}{t_0^2} + \frac{1}{2} \alpha \\
\frac{k_0}{t_0^2} &= \frac{1}{2} \alpha \\
t_0 &= \sqrt{\frac{2k_0}{\alpha}}.
\end{aligned}$$

With this value of $t_0$, we can compute the expected time for a computation that would take time $T$ with no checkpoints and no failures. That time is

$$\begin{aligned}
E_T &= T \left( 1 + \frac{k_0}{t_0} + \alpha \left( k_1 + \frac{1}{2} t_0 \right) \right) \\
&= T \left( 1 + \alpha k_1 + \sqrt{2 \alpha k_0} \right).
\end{aligned}$$

For small probability of failure $\alpha$ and relatively small costs of creating ($k_0$) and restoring ($k_1$) a checkpoint, the added cost of using checkpoints is quite modest. This result, combined with the modularity of checkpoint solutions (i.e., they do not require changes to the solution algorithm and can be implemented as separate modules in the the program source), helps explain the popularity o checkpoints.

Of course, the cost of saving and restoring a checkpoint must be relatively small. Since checkpoints must be saved to persistent storage that is not affected by a failure of one of the computing elements, the checkpoint data is typically saved to a (parallel) file system. Thus, the practicality of checkpointing is related to the performance of parallel I/O. MPI provides excellent facilities for performing I/O, including support for output that is in a canonical form and can be used to restart the computation on a different number of processors. The cost of restarting a computation ($k_1$) also need not be large, though in some implementations of MPI, startup time is nonnegligible. This part of the problem is best addressed through improved techniques such as those described in [3].

So far, we have ignored the issue of who is responsible for creating the checkpoint. Two major choices exist: user-directed and system-directed checkpointing. In user-directed checkpointing, the application programmer forms the checkpoint, writing out any data that will be needed to restart the application. This task is often relatively easy, particularly with well-structured applications. It has two drawbacks, however. First, the user is responsible for ensuring that all data is saved. Second, the checkpoints must be taken at particular points in the program, typically when no messages are in transit between processes. Doing so can be difficult for a program that does not have a simple iterative structure or that wishes to create a checkpoint only when necessary, such as when a failing component is detected. Unfortunately, system-directed checkpointing is much harder to implement because so much of a process's state is scattered throughout a parallel computer. This state can include messages that are in flight between processes and data in kernel memory buffers. Some work in this area was done for earlier message-passing systems [9, 10]. Because of the difficulty in extracting all of the necessary state, however, we recommend using user-directed checkpointing.

For user-directed checkpointing, source code transformation tools based on compiler technology can help identify both what data to checkpoint and what data need not be saved in a particular checkpoint (because that data was saved in a previous checkpoint and has not changed). In addition, such tools can ensure that all of the relevant data is saved and that the data is properly restored.

### 5.2 Using Intercommunicators

A fundamental concept in MPI is the *communicator*, a distributed object that supports both collective and point-to-point communication. Because of collective operations, the failure of any one process in a communicator affects all processes in the communicator, even those that are not in direct communication with the failed process. This factor contributes to the fragility of programs that use `MPI_COMM_WORLD` as their only communicator.

In contrast, in non-MPI client-server programs, the failure of a client has no significant effect on the server, which can continue to service other clients. What makes this structure robust is that all communication takes place in a two-party context, in which one party can easily recognize that the other party has failed and cease communicating with it. Moreover, each party can easily keep track of the state held by the other party.

How can we structure MPI programs so that they have this same type of survivability? The answer is to use the MPI structure that corresponds to two parties: the *intercommunicator*. Intercommunicators contain two groups of processes, and all communication occurs between processes in one group and processes in the other group.

Let us consider how we might use intercommunicators to make a common type of MPI application fault tolerant. Manager/worker algorithms are used in a wide variety of areas, from DNA sequence matching to graphics rendering to searching for extraterrestrial intelligence. In these algorithms, a manager process keeps track of a pool of tasks and dispatches them to working processes for completion. Workers return results to the manager, simultaneously requesting a new task. This arrangement is suitable for a fault-tolerant structure for two reasons:

– The worker processes hold a very small amount of state at a time, consisting of only the current task specification and the work that has been done on the task. The manager can keep a copy of the task specification and simply assign it to another worker if necessary.
– The communication structure is all two-party, between a worker and the manager. No collective communication is needed, and the workers do not communicate with one another.

A sample manager/worker example can be found in [8]. Figure 1 shows how to set up an MPI manager/worker program for fault tolerance, while the worker part of the algorithm is shown in Figure 2. We show here only the relevant MPI calls involved in setting up the intercommunicator structure and replacing the error handlers. A variety of data structures can be used for managing the task pool. We envision that for fault tolerance purposes, a task that has been sent to a worker is retained by the manager in an "in-progress" list, to keep the essential state of the worker that is working on it. In this way the manager can easily restart a task if the associated worker fails. If a process fails, as indicated by a non-success return code from a communication operation on one of the intercommunicators, the manager marks the communicator as invalid and does not use it again.

This program may not work on every implementation of MPI. The MPI implementation must be able to return a non-success return code in the case of a communication failure such as an aborted process or failed network link.

Note that in this example, we use only a single MPI process for the manager and for each worker, mimicking the non-MPI client-server structure. MPI intercommunicators can, however, contain arbitrary numbers of processes in both groups, so both the manager and the workers can be parallel programs in their own right; we can still use the same structure as described above, although setting up the intercommunicators will be a little more complicated.

We also note that our example is an MPI-1 program. If our implementation of MPI supports the dynamic process management part of MPI-2, then two changes can be considered:

– We can start the manager process by itself and use `MPI_Comm_spawn` to create the workers. `MPI_Comm_spawn` returns an intercommunicator, which can be used exactly as in the example above. In the parallel worker case, this will be a simpler way to form the intercommunicators, since one `MPI_Comm_spawn` can create multiple workers, all in the remote group of a single intercommunicator.

```c
#include "mpi.h"
#define MAX_WORKERS  1000
#define IC_CREATE_TAG 100

int main( int argc, char *argv[] )
{
    int i, myrank, origsize, currsize;
    MPI_Comm worker_comm[MAX_WORKERS];

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    MPI_Comm_size( MPI_COMM_WORLD, &origsize );
    currsize = origsize;

    /* create intercommunicators and set error handlers */
    for ( i = 1; i < currsize; i++ ) {
        MPI_Intercomm_create( MPI_COMM_SELF, 0,
                              MPI_COMM_WORLD, i,
                              IC_CREATE_TAG, &worker_comm[i-1]);
        MPI_Comm_set_errhandler( worker_comm[i-1], MPI_ERRORS_RETURN );
    }

    /* set up three lists of task descriptors:
     *   not done
     *   in progress
     *   done */

    /* manager part of manager worker algorithm:
     * when send or receive fails, mark intercommunicator as dead,
     * keep task in in-progress list, send to next free worker  */

    /* work until both not done and in progress lists are empty */

    for ( i = 1; i < currsize; i++ )
        MPI_Comm_free( &worker_comm[i-1] );
    MPI_Finalize( );
    exit( 0 );
}
```

**Fig. 1.** Manager part of fault-tolerant manager worker program

```
#include "mpi.h"
#define IC_CREATE_TAG 100

int main( int argc, char *argv[] )
{
    int i, myrank, size;
    MPI_Comm manager_comm;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    /* create intercommunicators for communication with manager
     * only */

    MPI_Intercomm_create( MPI_COMM_SELF, 0,
                          MPI_COMM_WORLD, 0,
                          IC_CREATE_TAG, &manager_comm );
    MPI_Comm_set_errhandler( manager_comm, MPI_ERRORS_RETURN );

    /* worker part of manager worker algorithm
     * when send or receive fails, mark intercommunicator as dead
     * and exit, since contact with manager is lost */

    MPI_Comm_free( &manager_comm );
    MPI_Finalize( );
    exit( 0 );
}
```

**Fig. 2.** Worker part of fault-tolerant manager worker program

– When a worker dies, we can use `MPI_Comm_spawn` to replace it and continue processing with no fewer workers than before.

The parallel workers can be optimized. The processes of a parallel worker may communicate with one another using an ordinary intracommunicator and utilize collective operations. The fault tolerance in this situation resides in the overall manager/worker structure. Thus, MPI's ability to attach different error handlers to different communicators allows us to provide different levels of fault tolerance to different parts of the program's structure.

### 5.3 Modifying MPI Semantics

Another approach to fault tolerance in MPI programs is to modify the semantics of certain MPI objects and functions in order to make it possible to write fault-tolerant programs for a wider set of algorithms than those considered in the preceding section, using existing MPI objects that contain more state and MPI functions that have a slightly different semantics from that defined in the MPI Standard. For example, in the approach taken in [4], a communicator can enter a state in which some ranks are defined and not others; moreover, a process's rank in a communicator may change, which cannot happen in a standard-conforming MPI implementation. The behavior of collective operations on a communicator in a non-standard state changes from the standard.

We believe that this approach, while intriguing as a way to experiment with fault-recovery algorithms, sacrifices too much in the area of time-tested semantics of MPI objects and functions to be realistic for writing production applications. For example, MPI objects have properties that the object model normally guarantees to be constant, such as the number of processes in a communicator and a process's rank in it. These properties may be used by the program in nontrivial ways: data may be decomposed according to a communicator's size, and the assignment of part of the data to a given process may be calculated by using its rank.

In the next section we propose an approach that is in some ways more limited but is more consistent with existing MPI programming style.

### 5.4 Extending MPI

One of the reasons for considering a change to the semantics of MPI is to address some of the real or perceived difficulties of using MPI communicators when processes may fail. Consider the case shown in Figure 3. In MPI, it is difficult (but not impossible) to construct the communicator consisting of the two indicated processes. If the manager group has suffered the failure of a process, it is even more difficult to construct the new communicator, because of the collective semantics of communicator construction in MPI.

Rather than modifying existing semantics, one can envision defining extensions to MPI that have semantics that support the writing of fault-tolerant programs but are consistent with all existing MPI semantics.

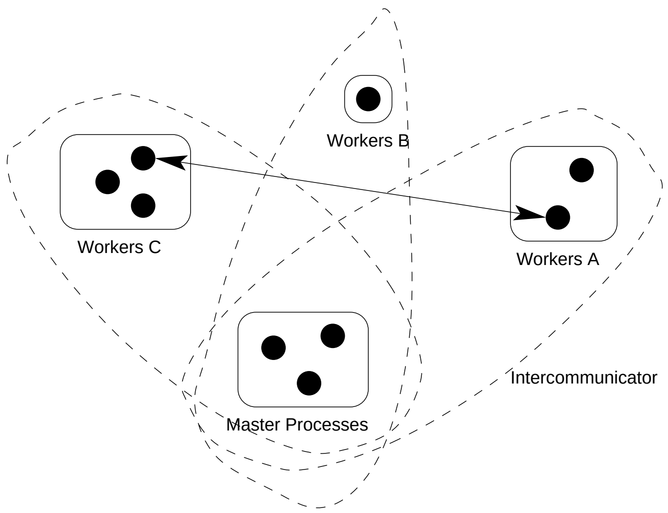


**Fig. 3.** Example manager-worker application with three intercommunicators connnecting separate groups of worker processes with a manager group. Consider constructing the communicator indicated by the arrow-headed line consisting of one process in group A and group C.

The key idea here is to encapsulate the capabilities we used in Section 5.2, where instead of using `MPI_COMM_WORLD` we based our communication on a local array of two-party connections. If this were incorporated into MPI, what might it look like? We will use the `MPE` prefix ("E" for "extension") to indicate that these are not MPI objects or functions. They could be added to an existing implementation and co-exist with standard MPI functions.

The central new object in this approach is the `MPE_Process_array` object, which behaves somewhat like the array of intercommunicators in the example of Section 5.2. A process array plays the role of communicator in communication operations but diffs in several ways. The properties of a process array include the following:

– Each element of the process array specifies a particular MPI process.
– A process array may grow or shrink in length (the number of members), but once an element is defined, it is fixed. In other words, the $i^{th}$ element of a process array always specifies the same process.
– If a process fails, the corresponding entry becomes null.
– Arrays will have associated contexts, like a communicator.
– No collective operations will be defined for process arrays.
– Arrays can have attached error handlers.
– Separate, new MPE send and receive operations will be defined.

Process arrays can be implemented on some systems by using `MPI_Join`. Process arrays provide a mechanism that does not, however, require a separate socket between the processes.

This mechanism is reminiscent of the `BLANK` option for the modified communicators of [4]. It can be used to develop programs that are fault tolerant, as in the manager/worker example, but have a more complex structure, for example one in which the workers communicate with one another, as in Figure 3.

## 6 Summary

In this paper we have discussed the meaning of fault tolerance as a program property that ensures survival of sufficient state for continuing the program. We have surveyed what the MPI Standard provides in the way of support for writing fault-tolerant programs. We have considered several approaches to doing so, and we have demonstrated how one can write fault-tolerant MPI programs.

## References


1. R. Batchu, J. Neelamegam, Z. Dui, M. Beddhua, A. Skjellum, Y. Dandass, and M. Apte. MPI/FT (TM): 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, Melbourne*, 2001.
2. George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cedile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vencent Neri, and Anton Selikhov. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Proceedings of SC 2002*. IEEE, 2002.
3. R. Butler, W. Gropp, and E. Lusk. Components and interfaces of a process management system for parallel programs. *Parallel Computing*, 27:1417–1429, 2001.
4. G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1495, October 2001.
5. Graham Fagg and Jack Dongarra. Fault-tolerant MPI: Supporting dynamic applications in a dynamic world. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virutal Machine and Message Passing Interface*, number 1908 in Springer Lecture Notes in Computer Science, pages 346–353, 2000. 7th European PVM/MPI Users' Group Meeting.
6. Al Geist and Christian Engelmann. Development of naturally fault tolerant algorithms for computing on 100,000 processors, 2002.
7. J. Gray and A. Reuter. *Transaction Processing*. Morgan Kaufmann Publishers, San Mateo (CA), USA, 1993.
8. William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface,* 2nd edition. MIT Press, Cambridge, MA, 1999.
9. Kai Li, Jeffrey F. Naughton, and James S. Plank. An Efficient Checkpointing Method for Multicomputers with Wormhole Routing. *International Journal of Parallel Processing*, 20(3):150–180, June 1992.
10. Kai Li, Jeffrey F. Naughton, and James S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):874–879, August 1994.

11. Soulla Louca, Neophytos Neophytou, Arianos Lachanas, and Paraskevas Evrepidou. MPI-FT: Portable fault tolerance scheme for MPI. *Parallel Processing Letters*, 10(4):371–382, 2000.
12. Message Passing Interface Forum. MPI: A Message-Passing Interface standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.
13. Message Passing Interface Forum. The MPI message-passing interface standard. `http://www.mpi-forum.org`, May 1995.
14. Sririam Rao, Lorenzo Alvisi, and Harrick M. Vin. Egida: an extensible tookit for low-overhead fault-tolerance. In *Symposium on Fault-Tolerant Computing*, pages 48–55, 1999.
15. Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI—The Complete Reference: Volume 1, The MPI Core,* 2nd edition. MIT Press, Cambridge, MA, 1998.