Process Fault Tolerance: Semantics, Design and Applications for High Performance Computing

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

International Journal of High Performance Computing Applications 2005 19: 465
DOI: 10.1177/1094342005056137

PROCESS FAULT TOLERANCE: SEMANTICS, DESIGN AND APPLICATIONS FOR HIGH PERFORMANCE COMPUTING

Graham E. Fagg[1]
Edgar Gabriel[2]
Zizhong Chen[1]
Thara Angskun[1]
George Bosilca[1]
Jelena Pjesivac-Grbovic[1]
Jack J. Dongarra[1]

Abstract

With increasing numbers of processors on current machines, the probability for node or link failures is also increasing. Therefore, application-level fault tolerance is becoming more of an important issue for both end-users and the institutions running the machines. In this paper we present the semantics of a fault-tolerant version of the message passing interface (MPI), the de-facto standard for communication in scientific applications, which gives applications the possibility to recover from a node or link error and continue execution in a well-defined way. We present the architecture of fault-tolerant MPI, an implementation of MPI using the semantics presented above as well as benchmark results with various applications. An example of a fault-tolerant parallel equation solver, performance results as well as the time for recovering from a process failure are furthermore detailed.

Key words: Parallel computing, fault tolerance, MPI and message passing

The International Journal of High Performance Computing Applications, Volume 19, No. 4, Winter 2005, pp. 465–477
DOI: 10.1177/1094342005056137

1 Introduction

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 thousands 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. Based on the current experiences with the high-end machines, it can be concluded, that a 100,000-processor machine will experience a processor failure every few minutes (Geist and Engelmann, 2005). While on earlier massively parallel processing systems (MPPs) crashing nodes often led 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 in the hardware as well as on the level of system software.

Current parallel programming paradigms for high performance computing systems mainly rely on message passing, in particular the message passing interface (MPI; MPI Forum, 1995) 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 means 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 run-time. MPI provides two options for handling failures. The first option, which is also the default mode of MPI, is to immediately abort the application. The second option is just slightly more flexible, handing the control back to the user application without guaranteeing that any further communication can occur. The latter mode's purpose is to mainly give an application the option to per-

[1]INNOVATIVE COMPUTING LABORATORY, COMPUTER SCIENCE DEPARTMENT UNIVERSITY OF TENNESSEE, KNOXVILLE, TN 37996-3450, USA, (FAGG@CS.UTK.EDU)

[2]HIGH PERFORMANCE COMPUTING CENTER STUTTGART, UNIVERSITY OF STUTTGART, D-70550 STUTTGART, GERMANY, AND INNOVATIVE COMPUTING LABORATORY, COMPUTER SCIENCE DEPARTMENT UNIVERSITY OF TENNESSEE, KNOXVILLE, TN 37996-3450, USA

form local operations before exiting, e.g. closing all files or writing a local check-point.

Summarizing the findings of the previous paragraphs, there is a discrepancy between the capabilities of current high performance computing systems and the most widely used parallel programming paradigm. While the robustness of machines is improving (hardware, network, operating systems, 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, check-point/restart, has performance and conceptual limitations. In fact, one of the main reasons many research groups prefer the parallel virtual machine (PVM; Geist et al., 1995) communication library to MPI is its capability to handle process failures.

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

- a specification proposing extensions to the MPI for handling process fault tolerance;
- an implementation of this specification based on the HARNESS framework (Beck et al., 1999);
- numerous application scenarios showing the feasibility of the specification for scientific, high performance computing.

The rest of the document is organized as follows. In Section 2 we present a summary of the fault-tolerant (FT) MPI specification as well as the architecture of the library and some implementation details. In Section 3 we compare the point-to-point performance of FT-MPI to those achieved with some popular public-domain MPI libraries. In Section 4 we use the Parallel Spectral Transform Shallow Water Model benchmark to further examine the overall performance of FT-MPI for numeric applications including the impacts of resource allocation via VAMPIR tracing of message transfers. In Section 5 we describe two applications that exploit the fault-tolerant features offered by FT-MPI: a master–slave framework and a preconditioned conjugate gradient solver. Finally, in Section 6 we summarize the paper and present the ongoing work.

1.1 RELATED WORK

The methods supported by various projects can be split into two classes: those supporting check-point/roll-back technologies, and those using replication techniques. The first method attempted to make MPI applications fault-tolerant through the use of check-pointing and roll-back. Co-Check MPI (Stellner, 1996) from the Technical University of Munich was the first MPI implementation built that used the Condor library for check-pointing an entire MPI application. Another system that also uses check-pointing but at a much lower level is StarFish MPI (Agbaria and Friedman, 1999). Unlike Co-Check MPI, Starfish MPI uses its own distributed system to provide built in check-pointing. LAM/MPI also supports system level check-pointing (Sankaran et al., 2003) with automatic roll-back of applications. Like Co-Check MPI, LAM/MPI is synchronous and completely transparent to the application, but relies on a third-party library to perform the actual check-point, which is currently the Berkeley Lab BLCR kernel-level process check-point library. The interaction between the check-point library and LAM/MPI is through a well-defined check-point/restart interface that in theory allows any check-point library to be used, further allowing LAM/MPI to be easily upgraded as newer check-point libraries become available.

MPICH-V (Bosilca et al., 2002) from Universite de Paris Sud, France is a mix of uncoordinated check-pointing and distributed message logging. The message logging is pessimistic, meaning that it guarantees a consistent state can be reached from any local set of process check-points at the cost of increased message logging. MPICH-V uses multiple message storage (observers) known as channel memories (CMs) to provide message logging. Process-level check-pointing is handled by multiple servers known as check-point servers (CSs). The distributed nature of the check-pointing and message logging allows the system to scale, depending on the number of spare nodes available to act as CM and CS servers.

LA-MPI (Graham et al., 2002) is a fault-tolerant version of MPI from the Los Alamos National Laboratory. Its main target is not to handle process failures, but to provide reliable message delivery between processes in presence of bus, networking cards, and wire-transmission errors. To achieve this goal, the communication layer is split into two parts; a memory and message management layer, and a send and receive layer. The first is responsible for choosing a different route in case the send and receive layer reports an error, while the message management layer is retransmitting lost packets.

MPI/FT (Batchu et al., 2001) provides fault tolerance by introducing a central co-coordinator and/or replicating MPI processes. Using these techniques, the library can detect erroneous messages by introducing a voting algorithm among the replicas and then can survive process failures. The drawback, however, is increased resource requirements and partial performance degradation.

FT-MPI, in common with non-message-logging check-point systems (such as LAM/MPI), can have much lower (or zero) overheads in terms of communications when no errors occur. These benefits do however have consequences. An application using FT-MPI has to be designed to take advantage of its fault-tolerant features, as shown in the next section, although this extra work can be trivial

depending on the structure of the application. If an application needs a high level of fault tolerance where node loss would equal data loss, then the application has to be designed to perform some level of user-directed check-pointing. Application level check-pointing using domain-specific knowledge can potentially produce much smaller check-points than system-level check-point libraries. An additional advantage of FT-MPI over many systems is that failure recovery can be performed at the user level and the entire application does not need to be rescheduled as with most process level check-pointing systems.

2 FT-MPI and HARNESS

In this section we present the extended semantics used by FT-MPI, the architecture of the library, and some details of the implementation. Furthermore, we present tools that support the application developer while using FT-MPI.

2.1 FT-MPI SEMANTICS

Implementing fault tolerance typically consists of three steps: failure detection, notification, and recovery. The FT-MPI specification does not make any assumptions about the first two steps except that the run-time environment discovers failures. In addition, 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 use of a special error code (MPI_ERROR_OTHER). As soon as an application process has received the notification of a death event through this error code, its general state is changed from *no failures* to *failure recognized*. While in this state, the process is only allowed to execute certain actions. These actions are dependent upon various parameters and are detailed later in the document.

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

The FT-MPI specification answers 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 recovery mode, the second by the communicator mode, and the third by the message mode and the collective communication mode. Each of the modes are described below.

2.1.1 FT-MPI Recovery modes 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 specific 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 most common mode used to date is the manual recovery mode. In this mode once the user's application has detected an error via the MPI_ERROR_OTHER return code they would start the recovery by calling MPI_COMM_DUP with both the values set to MPI_COMM_WORLD. In other MPI implementations this has no real meaning, but within FT-MPI it tells the library that the system level recovery can begin. The call is collective across all surviving MPI processes. Once completed, the returned new MPI_COMM_WORLD would be valid as discussed below.

It is possible to mix both the manual and automatic methods through the use MPI error handler routines. When FT-MPI wants to return the MPI_ERROR_OTHER code on detection of a node failure, it can instead trigger a handler routine which contains the MPI_COMM_DUP call. This method is not completely automatic as it still requires the users application to install an error handler.

2.1.2 FT-MPI Communicator and MPI Object states The status of MPI objects after the recovery operation is dependent upon whether they contain some global information or not. For MPI-1, the only objects containing global information are groups and communicators. These objects are invalidated during the recovery procedure. The objects available after MPI_Init, which are the communicators MPI_COMM_WORLD and MPI_COMM_SELF, are rebuilt by the library automatically.

Communicators and groups can have different formats after recovery operation. Failed processes may be replaced as in FTMPI_COMM_MODE_REBUILD. In cases where 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_

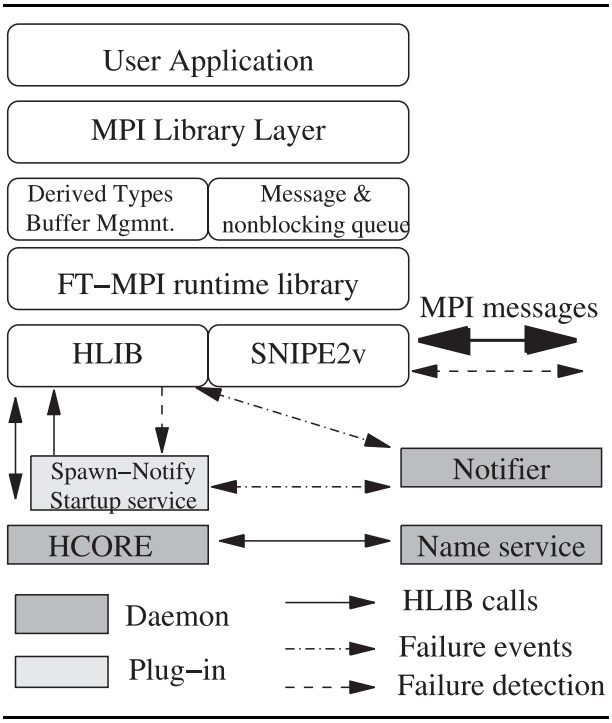

Fig. 1 A rchitecture of HARNESS and FT-MPI.

SHRINK). For both modes, a precise description of all MPI-1 functions are given in the FT-MPI specification.

2.1.3 FT-MPI Communication and Message Queue Handling Two modes are currently defined in the specification for how point-to-point messages and message queues are handled during and after failures. In the first mode, all messages in transit are canceled by the system. This mode is mainly useful for applications, which on 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 the roll-back. The preconditioned conjugate gradient solver shown in Section 5 details the usage of this communication mode.

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. All unmatched messages on queues are also kept so that non matched messages prior to a failure can latter be rematched after the failure. The exception to this is unmatched messages from failed processes, which are removed during the recovery operation. This mode requires that applications keep detailed information on the state of each process's message transfers, minimizing the rollback required.

Similar modes are available for collective operations, which can either be executed in an atomic or a non-atomic fashion. The master–slave example presented in Section 5 is an example of an application where no roll-back is necessary in case a process failure occurs.

2.2 ARCHITECTURE OF FT-MPI AND HARNESS

FT-MPI was built from the ground up as an independent MPI implementation as part of the Department of Energy Heterogeneous Adaptable Reconfigurable Networked SyStems (HARNESS) project (Beck et al., 1999). One of the aims of HARNESS was to provide a framework for distributed computing much like PVM (Geist et al., 1995). A major difference between PVM and HARNESS is the former's monolithic structure verses the latter's dynamic plug-in modularity. To provide users of HARNESS instant application support, both a PVM and a MPI plug-in were envisaged. As the HARNESS system itself was both dynamic and fault-tolerant (no single points of failure), it became possible to build a MPI plug-in with added capabilities such as dynamic process management and fault tolerance.

Figure 1 illustrates the overall structure of a user-level application running under the FT-MPI plug-in of the HARNESS system. In the following subsections we briefly outline the design of FT-MPI and its interaction with various HARNESS system components.

2.3 FT-MPI ARCHITECTURE

As shown in Figure 1, the FT-MPI system itself is built in a layered fashion. The uppermost layer handles the MPI-1.2 specification API and MPI objects. The next layer handles data conversion/marshaling (if needed), attribute and record storage, and various lists. Details of the highly tuned buffer management and derived data type handling can be found in Fagg et al. (2001). FT-MPI also implements a number of tuned MPI collective routines, which are further discussed in Vadhiyar et al. (2001). The lowest layer consists of the FT-MPI run-time library (FTRTL), which is responsible for interacting with the OS via the HARNESS user-level libraries (HLIBs). The FTRTL layer provides the facilities that allow for dynamic process management, system-level naming of MPI tasks, and message handling during the entire fault to recovery cycle. The HLIB layer interacts with the HARNESS system during startup, fault to recovery cycle, and shutdown phases of execution. The HLIB also provides the interfaces to the dynamic process management and redirection of application IO. *SNIPE2v*, a highly modified non-blocking asynchronous event driven version of the SNIPE (Fagg et al., 1999) library, provides the internode com-

munication of MPI message headers and data. SNIPE2v uses callbacks during the transmission of data so that the underlying network can control segmentation and conversion of complex data types and potentially overlap both. To simplify the design of the FTRTL, SNIPE only delivers whole messages atomically to the upper layers. During a recovery from failure, the non-event driven SNIPE library uses per channel flow control messages to indicate the current state of message handling (such as accepting connections, flushing messages, or in-recovery status).

It is important to note that the FTRTL shown in Figure 1 can receive notification of failures from both the point-to-point communications libraries as well as from the HARNESS layer. In the case of communication errors, the notification is usually started when the communication library detects a point-to-point message not being delivered to a failed party rather than the failed parties' OS layer detecting the failure. The FTRTL is responsible for notifying all tasks of errors as they occur by injecting notify messages into the send message queues ahead of user-level messages.

2.3.1 OS Support and the HARNESS G_HCORE The General HARNESS CORE (G_HCORE) is a daemon that provides a very lightweight infrastructure from which to build distributed systems. The capabilities of the G_HCORE are exploited via remote procedure calls (RPCs) as provided by the user-level library (HLIB). The core provides a number of very simple services that can be dynamically added to (Beck et al., 1999). The simplest service is the ability to load additional code in the form of a dynamic library (shared object) known as a plug-in and to make this available to either a remote process or directly to the core itself. Once the code is loaded it can be invoked using a number of different techniques such as the following.

- Direct invocation: the core calls the code as a function, or a program uses the core as a runtime library to load the function, which it then calls directly itself.
- Indirect invocation: the core loads the function and then handles requests to the function on behalf of the calling program, or it sets the function up as a separate service and advertises how to access the function.

An application built for HARNESS might not interact with the host OS directly but could instead install plug-ins that provide the required functionality. The handling of different OS capabilities would then be left to the plug-in developers, as is the case with FT-MPI.

2.3.2 G_HCORE Services for FT-MPI Services required by FT-MPI break down into two main categories.

- Spawn and notify service. This service is provided by a plug-in that allows remote processes to be initiated and then monitored. The service notifies other interested processes when a failure or exit of the invoked process occurs. The notify message is either sent directly to all other MPI tasks or via the FT-MPI Notifier daemon, which can provide additional diagnostic information if required.
- Naming services. These allocate unique identifiers in the distributed environment for tasks, daemons and services (which are uniquely addressable). The name service also provides temporary internal system (not application) state storage for use during MPI application startup and recovery, via a comprehensive record facility.

Currently, FT-MPI can be executed in one of two modes. The first mode is as a plug-in when executing as part of a HARNESS distributed virtual machine. A second mode is in a slightly lighter weight configuration with the spawn–notify service as a standalone daemon. This latter configuration loses the benefits of any other available HARNESS plug-ins, but is better suited for clusters that only execute MPI jobs. No matter which configuration is used, one name-service daemon, plus one either of the GHCORE daemon or one startup daemon per node, is needed for execution.

2.4 FT-MPI SYSTEM LEVEL RECOVERY ALGORITHM AND COSTS

The recovery method employed by FT-MPI is based on the construction of a consistent global state at a dynamically allocated leader node. The global state contains only the system-level information needed by FT-MPI to decide the MPI_COMM_WORLD communicator membership from which all other communicators are derived. This consists of ordered lists of Global IDentifiers (HARNESS GIDs) and communication channel connection information (such as TCP IP addresses and port numbers). The global state is not the application level state, which is discussed later in Section 5.

After the state is constructed at the leader node, it is distributed to all other nodes (peons) via an atomic broadcast operation based on a multiphase commit algorithm.

The recovery is designed to handle multiple recursive errors, including the failure of the leader node responsible for constructing the global state. Under this condition, an election state is entered where every node votes for themselves, and the first voter wins the election via an atomic swap operation on a leader record held by the HARNESS name service. Any other faults cause the leader node to restart the construction of the global state from the beginning. This process continues until the state is either com-

pletely lost (when all nodes already holding the previous verified state fail) or when everyone agrees with the atomic broadcast of the pending global state.

The cost of performing a system-level recovery is as follows.

- Synchronizing state and detecting faults: O(2N) messages.
- Respawning failed nodes and rechecking state and faults: O(2N) messages.
- Broadcasting the new pending global state, verifying reception: O(3N) messages.
- Broadcasting the acceptance of global state: O(N) messages.

The total cost of recovery from detection to acceptance of a new global state is O(8N) messages. The results detailed later in Section 5.2 currently use a linear topology for these messages leading to O(8N) cost, which is not acceptable for larger systems. Currently under test is a mixed fault-tolerant tree and ring topology, which together with the combining of several fault detection and broadcast stages will reduce the recovery cost to approximately O(3N) + O($3\log_2$N). This new method has also been designed for use in wide-area high-latency Grid-type environments.

3 Point-to-point Benchmark Results

In this section we would like to compare the point-to-point performance of FT-MPI with the performance achieved with the most widely used, non-fault-tolerant MPI implementations. These are MPICH (Gropp et al., 1996) using version 1.2.5 as well as the new beta-release of version 2 and LAM/MPI (Burns and Daoud, 1995) version 7. All tests were performed on a PC cluster consisting of 32 nodes, running a Linux 2.4 Kernel, each having two 2.4 GHz Pentium IV Xeon processors. The nodes are interconnected using Intel Gigabit Ethernet (82544EI) network cards.

For determining the communication latency and the achievable bandwidth, we used the latency test suite (Gabriel et al., 2003). The zero-byte latency measured in this test revealed LAM7 to have the best short-message performance, achieving a latency of 41.2 µs, followed by MPICH-2 with 43.6 µs. FT-MPI had in this test a latency of 44.5 µs, while MPICH 1.2.5 followed with 45.5 µs.

Figure 2 shows the achieved bandwidth with all communication libraries for large messages. FT-MPI achieved the highest bandwidth with a maximum of 66.5 MB/s. LAM7 and MPICH 2 have comparable results with 65.5 and 64.6 MB/s, respectively. The bandwidth achieved with MPICH 1.2.5 is slightly worse, having a maximum of 59.6 MB/s.

4 Performance Results with the Shallow Water Code Benchmark

While FT-MPI extends the syntax of the MPI specification, we expect that many of the end-users will use FT-MPI in the conventional, non-fault-tolerant way. Therefore, we evaluate in this section the performance of FT-MPI using the Parallel Spectral Transform Shallow Water Model (PSTSWM; Worley and Toonen, 1995) benchmark, and compare the performance results of FT-MPI with the results achieved with MPICH 1.2.5 and MPICH 2. LAM/MPI 7 is not included in this evaluation, since PSTSWM makes use of some optional Fortran MPI data types, which are currently not supported by LAM/MPI.

Included in the distribution of PSTSWM version 6.7.2 are several test cases and test data. Presenting the results achieved with all of these test cases would exceed the scope and the length of this paper, therefore we have picked three test cases, which we found representative from the problem size and performance behavior. All tests were executed with 32 processes using 16 nodes on the same PC cluster described in the previous section.

Figure 3 presents the results achieved for these three test cases. FT-MPI significantly outperformed MPICH 1.2.5 and MPICH-2 in these test cases. The reason turned out to be the process placement strategy of FT-MPI. FT-MPI distributes the processes block-wise, if the number of used processes does not match the number of available nodes. Thus, ranks 0 and 1 are located on the first node, ranks 2 and 3 on the second node, and so on. In contrast, both versions of MPICH distribute the processes in a cyclic manner, e.g. ranks 0 and 16 are on node 0, 1 and 17 on node 1.

Figure 4 shows a snapshot of the PSTSWM benchmark presenting the communication volume between each pair of nodes. This analysis reveals why the process distribution of FT-MPI could improve the performance compared to the other MPI libraries. The communication volume and the number of messages exchanged between neighboring processes (e.g. between rank 0 and rank 1) in this application is significantly higher than the overall data exchanged between other process pairs. Since the communication between processes within the same node is considerably faster than the communication between processes on different nodes, a larger number of messages could benefit from the faster communication inside a node using the block-wise process distribution.

In a second test, we forced FT-MPI to use a cyclic process distribution similar to MPICH. The results achieved in these tests are labeled as FT-MPI cyclic in Figure 3. In these measurements, FT-MPI and MPICH-2 are usually equally fast, but MPICH 1.2.5 remains slower than the other two MPI libraries. The overall conclusion of the last two sections are that the performance of FT-

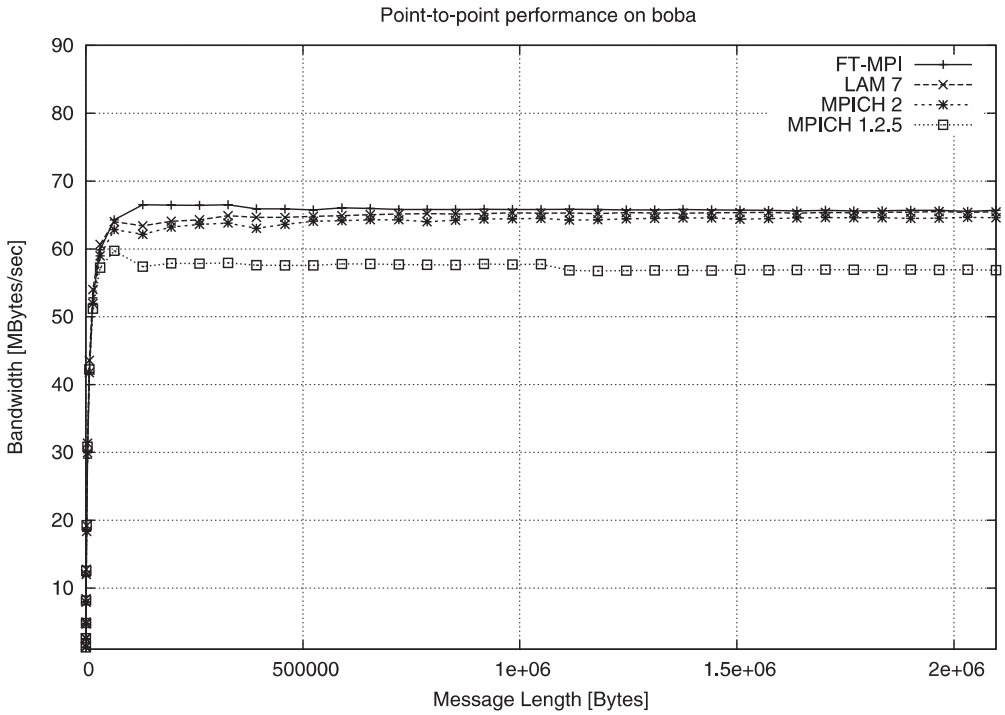

Fig. 2 Achieved bandwidths with FT-MPI, LAM 7, MPICH 1.2.5 and MPICH 2.

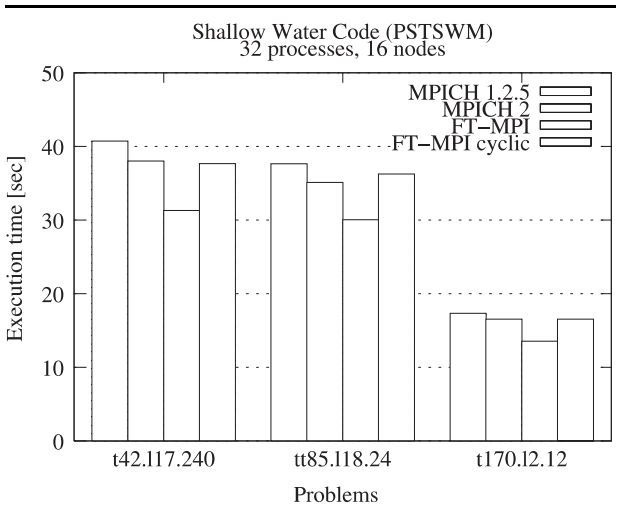

Fig. 3 Comparing the execution times for the PSTSWM benchmark for MPICH 1.2.5, MPICH 2, FT-MPI and FT-MPI using cyclic process distribution.

MPI is comparable to the current state-of-the-art public-domain MPI libraries. The extensions in the specification do not introduce a performance penalty per se in a non-fault-tolerant application.

5 Examples of Fault-tolerant Applications

Hand in hand with the development of FT-MPI, we also developed some example applications showing the usage of the fault-tolerant features of the library. In this section, we would like to present the relevant parts of fault-tolerant master–slave applications as well as a fault-tolerant version of a parallel equation solver.

5.1 A FRAMEWORK FOR A FAULT-TOLERANT MASTER–SLAVE APPLICATION

For many applications using a master–slave approach, fault tolerance can be achieved easily by adding a simple

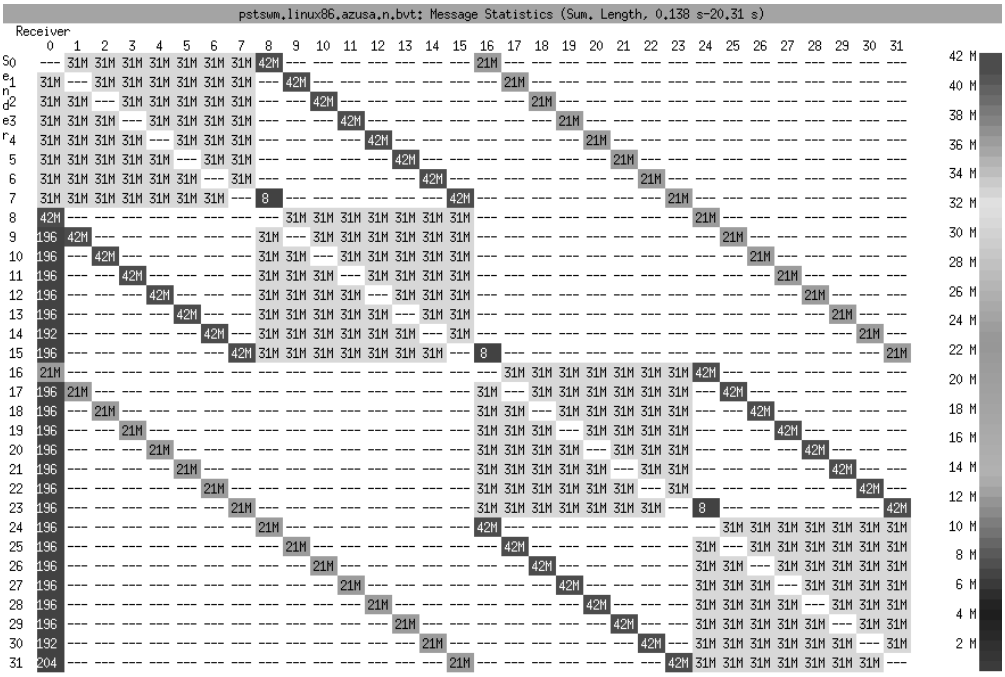

Fig. 4 Analysis of PSTSWM using VAMPIR. This figure shows the amount of data exchanged between each pair of processes during a typical run.

state model in the master process. The basic design follows that when a worker process dies, the master redistributes the work currently assigned to this process. When a master dies, the framework either restarts the work to be completed or aborts, depending on the respawn options available (i.e. if it can guarantee that the master remains rank zero or not).

The state model, as applied in our example, is shown in Figure 5.

The master maintains for every process their current state. This can be one of the following states.

- AVAILABLE: process is alive and no work has been assigned to him.
- WORKING: process is alive and work has been assigned to him.
- RECEIVED: process is alive and the result of its work has been received.
- FINISHED: process is alive and it has been notified that no more work will be sent to him.
- SEND FAILED: send operation to this process failed.
- RECV FAILED: the recv operation to this process failed.
- DEAD: this process is marked as dead.

Under regular conditions, the state of each process is changing from AVAILABLE to WORKING to RECEIVED and back to AVAILABLE. When an error occurs when distributing the work to the slaves, the state of the receiver process is changed to SEND FAILED. The send operation to this process could have failed for two reasons: first, the receiving process died, and secondly another process has failed. In both cases, all MPI operations called after the notification of the death-event of a process will return the specific MPI error code MPI_ERROR_OTHER. In the first case, the process is either marked as DEAD for the BLANK and SHRINK communicator mode or respawned and marked as AVAILABLE for the REBUILD mode. In the second case, the send operation has to be repeated, and the state of this process is then reset to its previous value. The situation is similar if an error occurs on the receive operation.

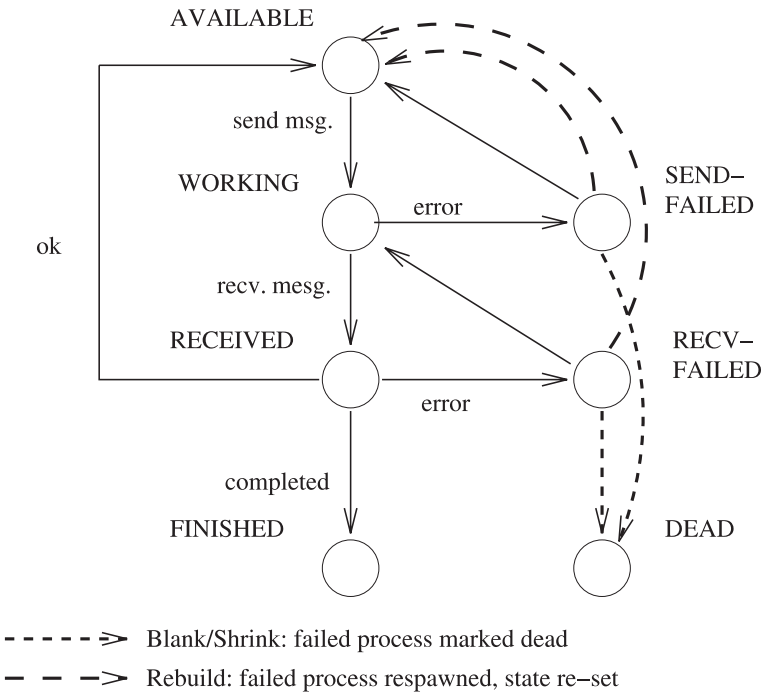

Fig. 5 Transition-state diagram of the fault-tolerant master–slave code.

The application can detect and handle failure events using two different methods, either the return code of every MPI function is checked, or the application makes use of MPI error handlers. The second method gives users the possibility to register a function with the MPI library, which is called, in case an error occurs. Thus, existing source code does not have to be extended by introduced detailed error-checking for each MPI function used.

The following extract of the source code of the master shows the most relevant pieces of the major working loop, including the registration of the error-handler as well as usage of the different states for each process. The transition state diagram is implemented in the routine .

```
/* Register error handler */
if ( master ) {
  MPI_Errhandler_create(recover,&errh);
  MPI_Errhandler_set ( comm, errh);
}

/* major master work loop */
do {

/* Distribute work */
for( proc=1; proc<maxproc; proc++)
    if( state[proc] == AVAILABLE ){
        MPI_Send(workid[proc],....);
        advance_state(proc);
     }

for( proc=1; proc<maxproc; proc++){
   /* Collect results */
   if( state[proc] == WORKING ){
      MPI_Recv(workid[proc], ....);
      advance_state(proc);
   }

   /* Perform global calculation */
   if( state[proc] == RECEIVED ) {
      workperformed += workid[proc];
      advance_state(proc);
   }
}

} while (all work is done);
```

The recovery algorithm invoked in case an error occurs consists of the following steps.

1. Re-instantiation of the MPI library and the run-time environment by calling a specific, predefined MPI function.
2. Determining how many processes have died and who has died.
3. Set the state of the failed processes to DEAD for the BLANK and SHRINK mode, respectively to AVAILABLE for the REBUILD mode.
4. Set the state of the communication partner in the Send or Recv operation when the error was detected to SEND_FAILED respectively RECV_FAILED.
5. Mark the piece of work, which was currently assigned to the failed processes as "not done".

The second point in the list is closely related to the problem of how a process can determine whether it has been part of the initial set of processes or whether it is a respawned processes. FT-MPI offers the user two possibilities to solve this issue. The first method, which is the fast solution, involves specific FT-MPI constants and attributes. In case a processes is a replacement for a failed process, the return value of MPI_Init will be set to a specific new FT-MPI constant (MPI_INIT_RESTARTED_PROCS). All surviving processes will have two additional MPI attributes set: the value of FTMPI_NUMFAILED_PROCS indicates how many processes have failed, while the value of FTMPI_ERR_FAILED is an error-code whose error-string contains the list of processes that have failed since the last error. This method is considered to be fast, since it does not involve any additional communication to determine these values.

The second possibility is that the application introduces a static variable. By comparing the value of this variable to the value on the other processes, the application can detect whether everyone has been newly started (in which case all processes will have the pre-initialized value) or whether a subset of processes have a different value, since each process modifies the value of this variable after the initial check. This second approach is more complex but is fully portable and can also be used with any other non-fault-tolerant MPI library.

5.2 A FAULT-TOLERANT PRECONDITIONED CONJUGATE GRADIENT SOLVER

In this section we would like to give an example of how fault tolerance can be achieved for a tightly coupled application, which is not using the master–slave paradigm. As an example, we implemented a parallel preconditioned conjugate gradient equation solver (PCG) in a fault-tolerant manner. The parallel application has been extended by two major points.

- *M* processes have been dedicated in the application to serve as in-memory check-point servers. Every 100 iterations, all processes calculate, using several MPI_Reduce operations a check-point of each relevant vector, which is then stored on the dedicated check-point processes.
- In case some of the processes die, the data of the respawned processes are recalculated using the local data on all other processes and the check-pointed vector. The matrix is not check-pointed in this application, since it is constant and not changing. Therefore, the respawned processes reread the matrix from the original input file.

The recovery algorithm makes use of the *longjmp* function of the C-standard. In the event that the return code of an MPI function indicates that an error has occurred, all processes jump to the recovery section in the code, perform the necessary operations, and continue the computation from the last consistent state of the application. The relevant section with respect to the recovery algorithm is shown in the source code below.

```
/* Mark entry point for recovery */
j = setjmp ( env );

1/* Execute recovery if necessary */
if ( state == RECOVER) {
   MPI_Comm_dup ( comm, &newcomm );
   comm = newcomm;
   ...
  /* do other operations */
  recover_data ( my_vector,.., &num_iter );

  /* reset state-variable */
  state = OK;
  }

/* major calculation loop */
do {
  ....

  rc = MPI_Send ( ...)
  if ( rc == MPI_ERR_OTHER ) {
    state = RECOVER;
    longjmp ( env, state );
  }

} while ( norm < errtol );
```

Table 1
Execution time for a 120-process FT-MPI CG application under various levels of burst failures.

Check-point servers/failures	Non-fault time (s)	Burst fault time (s)	Total extra/ratio (s)/(%)
0	6606.9	n/a	n/a
1	+7.6	19.5	27.1/0.4
2	+10.0	16.6	26.6/0.4
3	+12.8	16.9	29.7/0.45
4	+15.4	15.9	31.3/0.47
5	+18.2	14.6	32.8/0.5

The code is written such that any symmetric, positive definite matrix using the Boeing/Harwell format can be used for simulations. Table 1 gives some results of execution times for solving a system of linear equations using the fault-tolerant version of the solver for 120 processes with various numbers of concurrent failures. The solver executed 2000 iterations, making a check-point every 100 iterations, and the failures were forced on the 300th iteration by using the kill system call. The system of equations was represented by a sparse matrix 1.3 million elements square, with 51.4 million non-zero values.

The first column gives the number of check-point servers used. We also make this the number of concurrent failures. In other words, a system that has five check-point servers can also survive five concurrently failing application processes. The second column shows the application execution time when no faults occur. Thus, for the zero check-point row, this is the time for the non-check-pointing application. For all other rows this is the additional time for the application execution, i.e. the check-point overhead. The third column shows the additional time for the application to complete when a burst of failures occur once, i.e. the actual cost of a single exit-recovery phase. The number of failures is made equal to the number of check-point servers. (Thus, if we had five check-point servers we would kill five processes during the application run.) The last column shows the difference (or overhead) between a non-check-pointing application run and one surviving a single burst of failures with all overheads.

As Table 1 indicates, the additional overhead from check-pointing varies between 7.6 s for one check-point server to 18.2 s for five check-point servers. (The application performed 20 check-point iterations during the complete run.) The time for the application to recover from a single process failure was 19.5 s and reduced to 14.6 s for a burst of five processes failing. The overall overheads were between 0.4% and 0.5% of the non-check-pointing, no-failure execution. This is very competitive with other methods, although it did entail quite detailed tuning of the check-point methods used. Other applications with different memory usage patterns may produce very different results.

6 Conclusion and Outlook

In this paper we have presented the semantics of a fault-tolerant version of the MPI. FT-MPI is an implementation of this specification, supporting the full MPI-1.2 document as well as supporting extended functionality of a failure-recovery model. FT-MPI is, however, not an automatic check-point/recovery system. Instead it gives the application the potential to survive node or link failures, reorganize its communication and/or communicators, and continue from a well-defined point in the user application. Defining and implementing a consistent state in the application is the responsibility of the application developers.

Results with point-to-point benchmarks as well as with the PSTSWM benchmark show that the performance achieved with FT-MPI is comparable to other, non-fault-tolerant implementations of MPI, in some cases even better. The overhead introduced by the fault-tolerant features of the libraries are negligible.

Writing fault-tolerant applications usually requires some modifications to existing parallel applications. A state model for the development of master–slave applications has been presented as well as an example for a tightly coupled application, namely a parallel CG solver. The usage of error-handlers from the MPI specification greatly improves the readability and maintainability of fault-tolerant applications.

Current work focuses on improving the times for recovering from an error. Although for long-running applications the current system recovery times are just a marginal fraction of their overall execution time, we still think that there is ample room for further improvements in this area. More work will also be invested in the development of other templates to show how the fault-tolerant features of FT-MPI can be used by other classes of high performance computing applications.

Since FT-MPI was originally created, the HARNESS team at the University of Tennessee has become involved in the Open MPI (Gabriel et al., 2004) project. The project aims to combine the experience and features of four previous MPI projects: FT-MPI from the University of Tennessee, LA-MPI from Los Alamos National Laboratory, LAM/MPI from Indiana University, and PACX-MPI from the University of Stuttgart. Open MPI is expected to support both synchronous check-pointing (as available in LAM/MPI) and process fault tolerance from FT-MPI.

ACKNOWLEDGMENTS

This material is based upon work supported by the Department of Energy under Contract No. DE-FG02-02ER25536 and 8612-001-0449 through a subcontract from Rice University No. R7A827-792000. The National Science Foundation CISE Research Infrastructure program EIA-9972889 supported the infrastructure used in this work. The authors wish to say a special thank you to Scott Wells for his assistance during the final editing of this paper.

AUTHOR BIOGRAPHIES

Graham Fagg received his B.Sc. in computer science and cybernetics from the University of Reading (UK) in 1991 and a Ph.D. in computer science in 1998. From 1996 to 2001 he worked as a Senior Research Associate and then a Research Assistant Professor at the University of Tennessee. From 2001 to 2002 he was a visiting guest scientist at the High Performance Computing Center Stuttgart (HLRS). Currently, he is a Research Associate Professor at the University of Tennessee. His current research interests include distributed scheduling, resource management, performance prediction, benchmarking, cluster management tools, parallel and distributed IO and high-speed networking. He is currently involved in the development of a number of metacomputing and GRID middleware systems including SNIPE/2, MPI_Connect, HARNESS, a fault-tolerant MPI implementation (FT-MPI) and Open MPI. He is a member of the IEEE.

Edgar Gabriel is leading the working group on "Clusters and Distributed Units" at the High Performance Computing Center Stuttgart (HLRS), Germany. From January 2003 until August 2004 he was a Senior Research Associate and Adjunct Assistant Professor at the Innovative Computing Laboratory (ICL) at the University of Tennesse in Knoxville, USA. From January 2001 until December 2002 he was the leader of the working group "Parallel and Distributed Systems" at the HLRS. He holds a Ph.D. in mechanical engineering from the University of Stuttgart, Germany. His research interests are parallel and distributed computing, communication libraries, and fault tolerance.

Zizhong Chen received a B.Sc. in mathematics from Beijing Normal University in 1997 and an M.Sc. in computer science from the University of Tennessee, Knoxville in 2003. He is currently a Ph.D. candidate in computer science at the University of Tennessee, Knoxville. His research interests include high performance scientific computing, parallel and distributed system, fault tolerance and reliability, and numerical software for high performance computers.

Thara Angskun received his Bachelor and Master degrees in computer engineering from Kasetsart University, Thailand. Currently, he is a Ph.D. student and graduate research assistance at the Innovative Computing Laboratory, Department of Computer Science, University of Tennessee, Knoxville. His major research interests are in parallel and distributed environment, message passing, high performance computing, computer networking, cluster and grid computing. In 1999, he and his team integrated the 72-node Beowulf Cluster named PIRUN, which was the most powerful computer in Thailand at that time. He was a technical leader of an open source Beowulf cluster distribution project called OpenSCE. He was an SCE and HPAV division head, HPCNC, KU, Thailand. He is also a developer of several projects including KSIX, ACI, CAMETA and Harness/FT-MPI.

George Bosilca is a Senior Research Associate and Adjunct Assistant Professor at the Innovative Computing Laboratory (ICL) at the University of Tennessee in Knoxville, USA. Previously he was a Ph.D. student at the University of Paris XI, Orsay France in Parallel Architectures field, working in asynchronous parallel execution environments. Currently, he is involved in several projects involving parallel and distributed systems such as OVM, MPICH-V, and Harness/FT-MPI.

Jelena Pjesivac-Grbovic is a Ph.D. student and a Graduate Research Assistant at the Innovative Computing Laboratory (ICL), Department of Computer Science, University of Tennessee, Knoxville. She received Bachelor degrees in computer science and physics from Ramapo College, Mahwah, NJ in May 2003. Prior to coming to ICL, Jelena worked as Undergraduate Research Assistant in the Mathematical Modeling and Analysis group at Los Alamos National Laboratory, Los Alamos, NM. Her major research interest areas are parallel and distributed environments, high performance computing, cluster and grid computing, and computational biology.

Jack Dongarra received a B.Sc. in mathematics from Chicago State University in 1972 and an M.Sc. in computer science from the Illinois Institute of Technology in 1973. He received his Ph.D. in applied mathematics from the University of New Mexico in 1980. He worked at the Argonne National Laboratory until 1989, becoming a senior scientist. He now holds an appointment as University Distinguished Professor of Computer Science in the Computer Science Department at the University of Tennessee and holds the title of Distinguished Research Staff in the Computer Science and Mathematics Division at Oak Ridge National Laboratory (ORNL). Jack Dongarra specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He has contributed to the design and implementation of the following open source software packages and systems: EISPACK, LINPACK, the BLAS, LAPACK, ScaLAPACK, Netlib, PVM, MPI, NetSolve, Top500, ATLAS, and PAPI. He has published approximately 200 articles, papers, reports and technical memoranda and he is co-author of several books. He is a Fellow of the AAAS, ACM, and the IEEE and a member of the National Academy of Engineering.

References

Agbaria, A. and Friedman, R. 1999. Starfish: fault-tolerant dynamic MPI programs on clusters of workstations. *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, CA, August 3–6.

Batchu, R., Neelamegam, J., Cui, Z., Beddhua, M., Skjellum, A., Dandass, Y., and Apte, M. 2001. MPI/FT^{TM}: architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. *Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid*, Melbourne, Australia.

Beck, M. et al. 1999. HARNESS: a next generation distributed virtual machine. *Future Generation Computer Systems* 15(5–6):571–582.

Bosilca, G. et al. 2002. MPICH-V: Toward a scalable fault-tolerant MPI for volatile nodes. *Proceedings of Supercomputing*, Baltimore, MD, November.

Burns, G. and Daoud, R. 1995. Robust MPI message delivery through guaranteed resources. *MPI Developers Conference*, Notre Dame, IN, June.

Fagg, G. E., Moore, K., and Dongarra, J. J. 1999. Scalable networked information processing environment (SNIPE). *Future Generation Computing Systems* 15:571–582.

Fagg, G. E., Bukovsky, A., and Dongarra, J. J. 2001. HARNESS and fault-tolerant MPI. *Parallel Computing* 27: 1479–1496.

Gabriel, E., Fagg, G. E., and Dongarra, J. J. 2003. Evaluating the performance of MPI-2 dynamic communicators and one-sided communication. *Proceedings of the 10th European PVM/MPI Users' Group Meeting*, Venice, Italy, September 29–October 2.

Gabriel, E. et al. 2004. Open MPI: goals, concept, and design of a next generation MPI implementation. *Proceedings of the 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungry.

Geist, A. and Engelmann, C. 2005. Development of naturally fault-tolerant algorithms for computing on 100,000 processors. *Journal of Parallel and Distributed Computing* in press.

Geist, G., Kohl, J., Manchek, R., and Papadopoulous, P. 1995. New features of PVM 3.4 and beyond. *Recent Advances in Parallel Virtual Machine, 5th European PVM Users' Group Meeting, Lecture Notes in Computer Science*, Springer-Verlag, Berlin, pp. 1–10.

Graham, R. L., Choi, S-E., Daniel, D. J., Desai, N. N., Minnich, R. G., Rasmussen, C. E., Risinger, L. D., and Sukalski, M. W. 2002. A network-failure-tolerant message-passing system for terascale clusters. *ACM International Conference on Supercomputing*, New York, NY, June 22–26.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing* 22(6):789–828.

Louca, S., Neophytou, N., Lachanas, A., and Evripidou, P. 2000. MPI-FT: portable fault tolerance scheme for MPI. *Parallel Processing Letters* 10(4):371–382.

MPI Forum. 1995. *MPI: A Message Passing Interface Standard*, June, http://www.mpi-forum.org/.

MPI Forum. 1997. *MPI-2: Extensions to the Message Passing Interface*, July, http://www.mpi-forum.org/.

Sankaran, S., Squyres, J. M., Barrett, B., Lumsdaine, A., Duell, J., Hargrove, P., and Roman, E. 2003. The LAM/MPI checkpoint/restart framework: system-initiated checkpointing. *Proceedings of the LACSI Symposium*, Santa Fe, NM, October.

Stellner, G. 1996. Cocheck: checkpointing and process migration for MPI. *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, HI.

Vadhiyar, S. S., Fagg, G. E., and Dongarra, J. J. 2001. Performance modeling for self-adapting collective communications for MPI. *Proceedings of the LACSI Symphosium*, Santa Fe, NM, October 15–18.

Worley, P. H. and Toonen, B. 1995. *A Users's Guide to PSTSWM*, ORNL Technical Report ORNL/TM-12779, July.