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
Innovative Computing Laboratory, Computer Science Department,
University of Tennessee, 1122 Volunteer Blvd., Suite 413, Knoxville, TN 37996-3450, USA
{fagg, egabriel, zchen, angskun, bosilca, pjesa, dongarra}@cs.utk.edu

Abstract

With increasing numbers of processors on todays 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. This paper presents the semantics of a fault tolerant version of the Message Passing Interface, 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. The architecture of FT-MPI, an implementation of MPI using the semantics presented above as well as benchmark results with various applications are presented. 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.

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 thousand of processors. Machines comprising 100,000 processors are expected for the next years.

A critical issue of systems consisting of such large numbers of processors is the ability of the machine to deal with process failures. 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 [21]. While on earlier massively parallel processing systems (MPPs) crashing nodes often lead to a crash of the whole system, current architectures are more robust. Typically, the applications utilizing the failed processor will have to abort, the machine, as an entity is however not affected by the failure. This robustness has been the result of improvements at the hardware as well as on the level of system software.

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 has mainly the purpose to give an application the possibility to perform local operations before exiting, e.g. closing all files or writing a local checkpoint.

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 machines are improving their robustness (hardware, network, operating systems, file systems) the MPI specification does not leave room for fully exploiting the capabilities of the cur-

rent 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.

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

- A specification proposing extensions to the Message-Passing Interface for handling process fault-tolerance,
- An implementation of this specification based on the HARNESS framework,
- 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 Fault-Tolerant MPI specification as well as the architecture of the library and some implementation details. Section 3 compares the point-to-point performance of FT-MPI to those achieved with some popular public-domain MPI libraries, while section 4 uses the Parallel Spectral Shallow Water Code benchmark to classify the performance of FT-MPI. In section 5 we describe two applications which exploit the fault-tolerant features offered by FT-MPI: a master-slave framework and a preconditioned conjugate gradient solver. Finally, section 6 summarizes the paper and presents the ongoing work.

1.1 Related Work

The methods supported by various projects can be split into two classes: those supporting checkpoint/roll-back technologies, and those using replication techniques. The first method attempted to make MPI applications fault tolerant was through the use of check-pointing and roll back. Co-Check MPI [18] from the Technical University of Munich being 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 [4]. Unlike Co-Check MPI, Starfish MPI uses its own distributed system to provide built in check-pointing.

MPICH-V [7] from Universite´e de Paris Sud, France is a mix of uncoordinated check-pointing and distributed message logging. The message logging is pessimistic thus they guarantee that a consistent state can be reached from any local set of process checkpoints at the cost of increased message logging. MPICH- V uses multiple message storage (observers) known as Channel Memories (CM) to provide message logging. Process level check-pointing is handled by multiple servers known as Checkpoint Servers (CS). 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 [13] 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 one 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 [5] 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 can survive process-failures. The drawback however is increased resource requirements and partially performance degradation.

FT-MPI has much lower overheads compared to the above check-pointing and message replication systems, and thus much higher potential performance. 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. An additional advantage of FT-MPI over many systems is that check-pointing can be performed at the user level and the entire application does not need to be stopped and rescheduled as with most process level check-pointing systems.

2 FT-MPI and HARNESS

This section presents the extended semantics used by FT-MPI, the architecture of the library as well as some details of the implementation. Furthermore, we present tools which are supporting the application developer while using FT-MPI are also presented.

2.1 FT-MPI Semantics

Handling 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 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 invalidated during the recovery procedure. The objects available after MPI_Init, which are the communicators MPI_COMM_WORLD and MPI_COMM_SELF, are re-instantiated by the library automatically.

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

Furthermore, the specification clarifies the status of messages when errors occur. Two modes are currently defined in the specification. In the first mode, all messages in transit are canceled 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 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. This mode requires that the application keeps 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. 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 [6]. One of the aims of HARNESS was to provide a framework for distributed computing much like PVM [12] previously. A major difference between PVM and HARNESS is the formers 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), then 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, and HARNESS system. The following subsections 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 layering fashion. The upper most layer deals with the handling of the MPI-1.2 specification API and MPI objects. The next layer deals with 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 [9]. FT-MPI also implements a number of tuned MPI collective routines, which are further discussed in [19]. The lowest layer consists of the FT-MPI runtime library (FTRTL), which is responsible for interacting with the OS via the HARNESS user level libraries (HLIB). The FTRTL layer provides the facilities that allow for dynamic process management, system level naming of MPI tasks, message handling during the entire fault to recovery cycle. The HLIB layer interacts with HARNESS system during both 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. The SNIPE [10] library provides the inter-node communication of MPI message headers and data. To simplify the design of the FTRTL, SNIPE only delivers whole messages atomically to the upper layers. During a recovery from failure, SNIPE uses in channel system flow control messages to indicate the current state of message handling (such as accepting connections, flushing messages or in-recovery).

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 notify is usually started by the communication library detecting 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 [1]. 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 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:

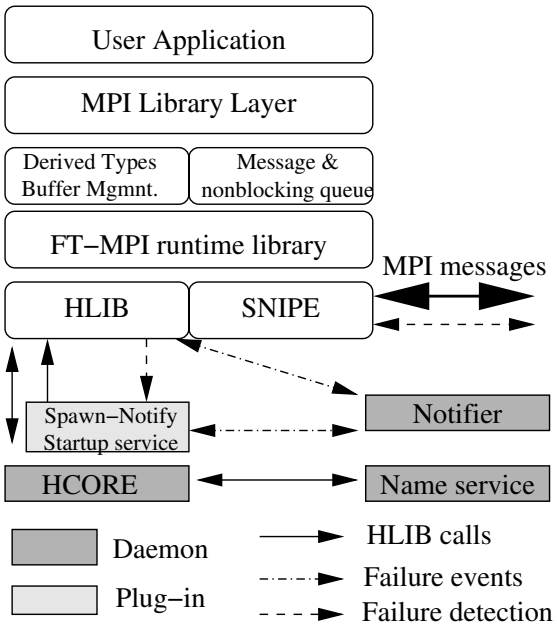

Figure 1. Architecture of HARNESS and FT-MPI

- 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 which 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 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. As the plug-in mode described above when executing as part of a HARNESS distributed virtual machine, or 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 is the bases for the MPI_COMM_WORLD communicator membership from which all other communicators are derived. After the state is constructed at this node it is distributed to all other nodes (*peons*) via an atomic broadcast operation based on a multi-phase 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 causes the leader node to restart the construction of the global state from the beginning. This process continues until the state is either completely 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.
- re-spawning 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).

3 Point-to-point benchmark results

In this section we would like to compare the point-to-point performance of FT-MPI to the performance achieved with the most widely used, non fault-tolerant MPI implementations. These are MPICH [14] using version 1.2.5 as well as the new beta-release of version 2, and LAM/MPI [8] version 7. All tests were performed on a PC-cluster consisting of 32 nodes, each having two 2.4 GHz Pentium IV Xeon processors. The nodes are connected by a Gigabit Ethernet network.

For determining the communication latency and the achievable bandwidth, we used the latency test suite [11]. 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 achieves in this test the best bandwidth with a maximum of 66.5 MB/s. LAM7 and MPICH 2 have comparable results with 65.5 MB/s 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) [20] benchmark, and compare the performance results of FT-MPI to the results achieved with MPICH 1.2.5

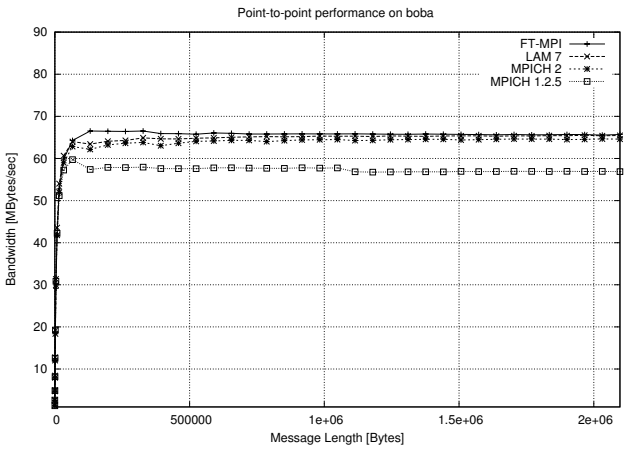

Figure 2. Achieved bandwidth with FT-MPI, LAM 7, MPICH 1.2.5 and MPICH 2

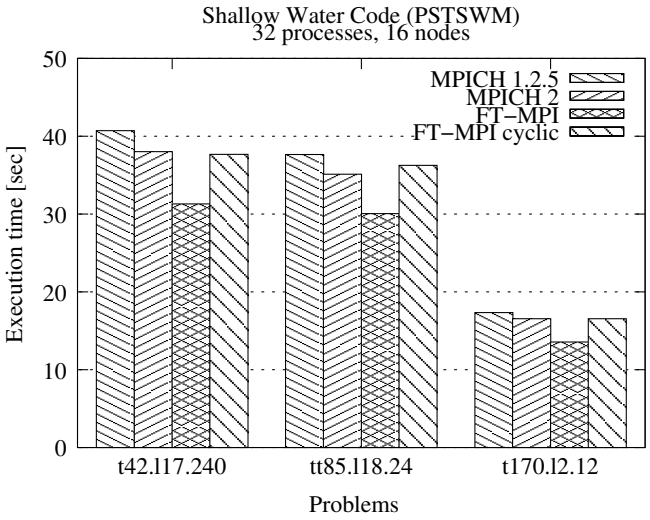

Figure 3. Comparing the execution times for the PSTSWM benchmark for MPICH 1.2.5, MPICH 2, FT-MPI and FT-MPI using cyclic process distribution.

and MPICH 2. LAM/MPI 7 is in contrary to the previous section not included in this evaluation, since PSTSWM makes use of some optional Fortran MPI-Datatypes, 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 outperformed MPICH 1.2.5 and MPICH-2 in these test-cases significantly. 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, the ranks 0 and 1 or located on the first node, ranks 2 and 3 on the second node, etc.. In contrary to that, both versions of MPICH distribute the processes in a cyclic manner, e.g. the 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) is in this application 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. For these measurements, FT-MPI and MPICH 2 are usually equally fast, 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-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 state model in the master process. The basic idea is, that

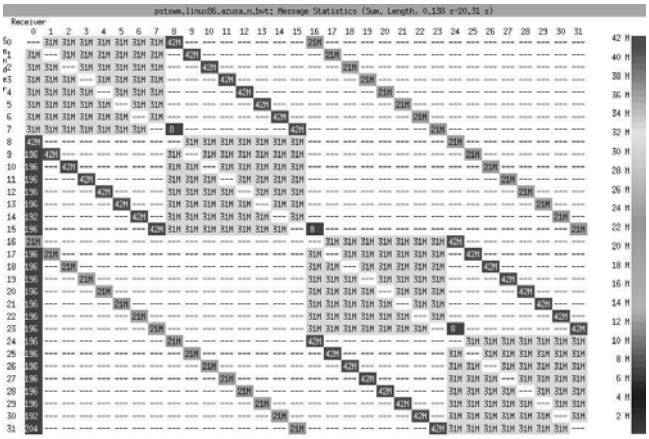

Figure 4. Analysis of PSTSWM using VAMPIR. This figure shows the amount of data exchanged between each pair of processes during a typical run.

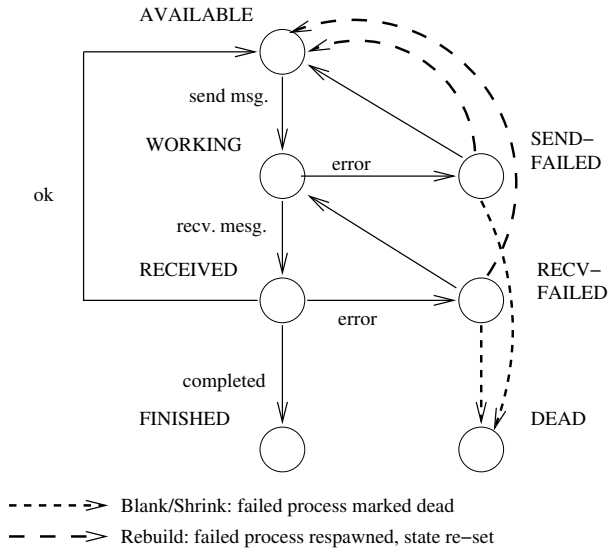

Figure 5. Transition-state diagram of the fault-tolerant master-slave code

when a worker process dies, the master redistributes the work currently assigned to this process. The state model as applied in our example, is shown in figure 5.

The master maintains for every process its 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. In case 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 due to two reasons: first, the receiving process died, and second 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 mentioned in section 2. In the first case, the process is either marked as DEAD for the BLANK and SHRINK communicator mode, or re-spawned and marked as AVAILABLE for the REBUILD mode. In the second case, the Send operation has to be repeated, and therefore the state of this process is re-set to its previous value. The situation is similar if an error occurs on the receive operation.

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 mode 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 peaces of the major working loop, including the registration of the error-handler as well as usage of the different state's for each process. The transition state diagram is implemented in the routine `advance_state`.

```
/* 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 runtime 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, 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 is the fast solution, involves however proprietary 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 which 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 everybody 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 processes modifies the value of this variable after the initial check. This second approach is somewhat more complex, however, it 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:

- A process has been dedicated in the application to serve as an in-memory checkpoint server. Every 200 iterations, all processes calculate using several MPI_Reduce operations a checkpoint of each relevant vector, which is than stored on the dedicated checkpoint processes.
- In case one of the processes dies, the data of the respawned process is recalculated using the local data on all other processes and the checkpointed vector. The matrix is not checkpointed in this application, since it is constant and not changing. Therefore, the respawned processes rereads the matrix from the original input file.

The recovery algorithm makes use of the *longjmp* function of the C-standard. In case 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 );
```

```
/* 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 );
```

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. The first column is indicating the problem size by giving the number of non-zero entries in the matrix, the second column the number of processes used for the calculation (with the checkpoint process also shown as an addition) The third column contains the execution time required to achieve a solution with the required precision. Finally, the fourth column is showing the recovery time in both seconds and as a ratio of overall execution time in case of a computational processes dies.

Problem size	No. of procs.	Exec. time [sec]	Recovery time/ratio [sec]/[%]
4,054	4+1	5	1.32/26.4
428,650	8+1	189	3.34/1.76
2,677,324	16+1	1162	11.37/0.97

Table 1. Execution time of various problem sizes with FT-MPI, and the recovery time for the according number of processes

As table 1 indicates, recovering from an exit-event of a process takes between 1.3 seconds for 4 processes to 11.37 seconds for 16 processes. The recovery time itself can be split again into two major parts: the first part is the time spent in a multi-phase commit protocol between the processes, since FT-MPI is capable of recovering from additional death events recursively during the recovery phase. For 16 processes, this time is in our application scenario 3.5 out of 11.37 seconds. The second part of the recovery time consists of the recovery of the user-data within the application. While this time is negligible for small matrices and problem sizes, it is the dominant part for the biggest test case their. The re-reading of the matrix over NFS at the respawned process takes up to 7.5 seconds. Nevertheless, for large problem sizes, the overall recovery time from a process failure is less than one percent of the total execution time of the application. Therefore, the advantage for the end-user is, that despite the potential of having to deal with process failures, the overall execution time for the simulation in the case of failures are approximately the same as for the original execution without failures.

6 Conclusion and Outlook

In this paper we presented the semantics of a fault-tolerant version of the Message Passing Interface. 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 checkpoint/recovery system, instead it gives the application the possibility to survive node or link failures, re-organize its communication and/or communicators and continue from a well defined point in the users 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 requires usually 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. While for long-running applications even the current 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.

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.

References

[1] World Wide Web. `http://www.es.jamstec.go.jp/esc/eng/Press/index.html`.

[2] World Wide Web. `http://www.lanl.gov/projects/asci/`.

[3] World Wide Web. `http://www.research.ibm.com/bluegene/`.

[4] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *In 8th IEEE International Symposium on High Performance Distributed Computing*, 1999.

[5] R. Batchu, J. Neelamegam, Z. Cui, 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 *In Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid held in Melbourne, Australia.*, 2001.

[6] Beck, Dongarra, Fagg, Geist, Gray, Kohl, Migliardi, K. Moore, T. Moore, Papadopoulous, Scott, and Sunderam. HARNESS:a next generation distributed virtual machine. *Future Generation Computer Systems*, 15, 1999.

[7] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *SuperComputing*, Baltimore USA, Novembre 2002.

[8] G. Burns and R. Daoud. Robust MPI message delivery through guaranteed resources. In *MPI Developers Conference*, June 1995.

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

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

[11] E. Gabriel, G. E. Fagg, and J. J. Dongarra. Evaluating the performance of MPI-2 dynamic communicators and one-sided communication. Sept. 2003. 10th European PVM/MPI Users' Group Meeting.

[12] G. Geist, J. Kohl, R. Manchek, and P. Papadopoulous. New features of pvm 3.4 and beyond. In *Recent Advances in Parallel Virutal Machine*, Lecture Notes In Computer Science, pages 1–10. Springer, Sept. 1995. 5th European PVM Users' Group Meeting.

[13] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *ICS*. New York, USA, June. 22-26 2002.

[14] 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, Sept. 1996.

[15] S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. Mpi-ft: Portable fault tolerance scheme for mpi. In *Parallel Processing Letters, Vol. 10, No. 4, 371-382 , World Scientific Publishing Company.*, 2000.

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

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

[18] G. Stellner. Cocheck: Checkpointing and process migration for mpi. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.

[19] S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Performance modelling for self-adapting collective communications for MPI. In *LACSI Symphosium*. Springer, Eldorado Hotel, Santa Fe, NM, Oct. 15-18 2001.

[20] P. H. Worley and B. Toonen. *A Users's Guide to PSTSWM*, July 1995.

[21] A. Geist and C. Engelmann. *Development of Naturally Fault Tolerant Algorithms for Computing on 100,000 Processors*. Journal of Parallel and Distributed Computing, to be published.