Fault Tolerant Communication Library and Applications for High Performance Computing

Graham E. Fagg, Edgar Gabriel, Zizhon Chen,
Thara Angskun, George Bosilca, Antonin Bukovsky, and Jack J. Dongarra
Innovative Computing Laboratory, Computer Science Department,
University of Tennessee, Knoxville, TN, USA
{fagg, egabriel, 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 becomin 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

Application developers and end-users of high performance computing systems have today access to larger machines and more processors than ever before. Systems like the Earth Simulator [1], the ASCI-Q machines [2] or even more extremely the IBM Blue Gene machine [3] consist of thousands of processors. Additionally, not only the individual machines are getting bigger, but with the recently increased network capacities, users have access to higher number of machines and computing resources. Concurrently using several computing resources, often referred to as Grid- or Metacomputing, further increases the number of processors used in each single job as well as the overall number of jobs, which a user can launch.

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

The Message Passing Interface (MPI) [16, 17] is the defacto standard for the communication in scientific applications. However, MPI in its current specification gives the user no possibility to handle the situation mentioned above, where one or more processors are becoming unavailable during runtime. Currently, MPI gives the user the choice between two possibilities of how to handle a failure. The first possibility is the default mode, which is to immediately abort the application. The second possibility is to hand the control back to the user application (if possible) without guaranteeing, that any further communication can occur. The latter mode mainly has the purpose of giving the application the possibility to close all files properly, write maybe a per-process based checkpoint etc., before exiting the application.

This situation is however unsatisfactory. Not only are large numbers of CPU hours wasted and lost, but also in the case of very long running or security relevant applications this might not be an option at all. The importance of this problem can also be seen by the numerous efforts in this area , e.g. FT-MPI [9], MPI/FT [5], MPI-FT [15], MPICH-V [7], LA-MPI [13].

In this paper we would like to present the concept and the current status of FT-MPI, a fault-tolerant version of MPI developed at the University of Tennessee, Knoxville. Furthermore, we would like to give a detailed presentation of how to write a fault tolerant applications, using a master-slave approach as well as a parallel equation solver as examples. The structure of the paper is as follows. In section 2 we present the semantics, concept and architecture of FT-MPI. Section 3 focuses on the performance comparison of FT-MPI with other MPI libraries for point-to-point benchmarks, while section 4 does a similar performance comparison using the Parallel Spectral Transform Shallow Water Model (PSTSWM) benchmarks. In section 5 we describe various techniques for developing fault tolerant applications. Furthermore, the concept of a fault tolerant equation solver as well as execution and recovery times for various problem sizes are shown. Section 6 finally presents the current status of FT-MPI as well as the ongoing work in this area.

1.1 Related Work

The methods supported by various project 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 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 resubmitting lost packets or choosing a different route, in case the Send and Receive Layer reports an error.

MPI/FT [5] provides fault-tolerance by introducing a central co-ordinator 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.

The project closest to FT-MPI known to the author is the Implicit Fault Tolerance MPI project MPI-FT [15] by Paraskevas Evripidou of Cyprus University. This project supports several master-slave models where all communicators are built from grids that contain ’spare’ processes. These spare processes are utilized when there is a failure. To avoid loss of message data between the master and slaves, all messages are copied to an observer process, which can reproduce lost messages in the event of any failures. This system appears only to support SPMD style computation and has a high overhead for every message and considerable memory needs for the observer process for long running applications.

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. FT-MPI does allow for atomic communications much like Starfish, but unlike Starfish, the level of correctness can be varied on for individual communicators. This provides users the ability to fine tune for coherency or performance as system and application conditions dictate. 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 process level check-pointing.

2 Harness and FT-MPI

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

2.1 FT-MPI Semantics

Current semantics of MPI indicate that a failure of a MPI process or communication causes all communicators associated with them to become invalid. As the standard provides no method to reinstate them, we are left with the problem that this causes MPI_COMM_WORLD itself to become invalid and thus the entire MPI application will grid to a halt.

FT-MPI extends the MPI communicator states from valid, invalid to a range FT_OK, FT_DETECTED, FT_RECOVER, FT_RECOVERED, FT_FAILED. In essence this becomes OK, PROBLEM, FAILED, with the other states mainly of interest to the internal fault recovery algorithm of FT_MPI. Processes also have typical states of OK, FAILED which FT-MPI replaces with OK, Unavailable, Joining, Failed. The Unavailable state includes unknown, unreachable or "we have not voted to remove it yet" states. A communicator changes its state when either an MPI process changes its state, or a communication within that communicator fails for some reason.

On detecting a failure within a communicator, that communicator is marked as having a probable error. Immediately as this occurs the underlying system sends a state update to all other processes involved in that communicator. If the error was a communication error, not all communicators are forced to be updated, if it was a process exit then all communicators that include this process are changed. How the system behaves depends on the communicator failure mode chosen by the application. The mode has two parts, one for the communication behavior and one for the how the communicator reforms if at all..

2.1.1 Communicator and communication handling

Once a communicator has an error state it can only recover by rebuilding it, using a modified version of MPI_Comm_dup. Using this function the new communicator will follow the following semantics depending on its failure mode:

- SHRINK: The communicator is reduced so that the data structure is contiguous. The ranks of the processes are changed, forcing the application to recall MPI_COMM_RANK.
- BLANK: This is the same as SHRINK, except that the communicator can now contain gaps to be filled in later. Communicating with a gap will cause an invalid rank error. Note also that calling MPI_COMM_SIZE will return the extent of the communicator, not the number of valid processes within it.
- REBUILD: Most complex mode that forces the creation of new processes to fill any gaps until the size is the same as the extent. The new processes can either be places in to the empty ranks, or the communicator can be shrank and the remaining processes filled at the end. This is used for applications that require a certain size to execute as in power of two FFT solvers.
- ABORT: Is a mode which affects the application immediately an error is detected and forces a graceful abort. The user is unable to trap this. If the application need to avoid this they must set all communicators to one of the above communicator modes.

Communications within the communicator are controlled by a message mode for the communicator which can be either of:

1. NOP: No operations on error. I.e. no user level message operations are allowed and all simply return an error code. This is used to allow an application to return from any point in the code to a state where it can take appropriate action as soon as possible.
2. CONT: All communication that is NOT to the affected/failed node can continue as normal. Attempts to communicate with a failed node will return errors until the communicator state is reset.

2.1.2 Point to Point versus Collective correctness

Although collective operations pertain to point to point operations in most cases, extra care has been taken in implementing the collective operations so that if an error occurs during an operation, the result of the operation will still be the same as if there had been no error, or else the operation is aborted.

Broadcast, gather and all gather demonstrate this perfectly. In Broadcast even if there is a failure of a receiving node, the receiving nodes still receive the same data, i.e. the same end result for the surviving nodes. Gather and all-gather are different in that the result depends on if the problematic nodes sent data to the gatherer/root or not. In the case of gather, the root might or might not have gaps in the result. For the all2all operation, which typically uses a ring algorithm it is possible that some nodes may have complete information and others incomplete. Thus for operations that require multiple node input as in gather/reduce type operations any failure causes all nodes to return an error code, rather than possibly invalid data. Currently an addition flag controls how strict the above rule is enforced by utilizing an extra barrier call at the end of the collective call if required.

2.1.3 Application view

The library provides the application a possibility to recover from an error, restructure itself and continue with the execution. However, the application has to take some steps itself to handle an error properly. Two possibilities are offered by FT-MPI:

- The user discovers any errors from the return code of any MPI call, with a new fault indicated by MPI_ERR_OTHER. Details as to the nature and specifics of an error are available though the cached attributes interface in MPI.

- The user can register a new error-handler at the beginning of the simulation, which is than called by the MPI-library in case an error occurs. Using this mechanism, the user hardly needs to change any code.

The error-recovery function of the application has to perform two phases: first, all non-local information needs to be reestablished (e.g. all communicators derived from another communicator, which has an erroneous processes needs to be re-created). Second, the application needs to resume from a well defined state in the application.

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 an 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).

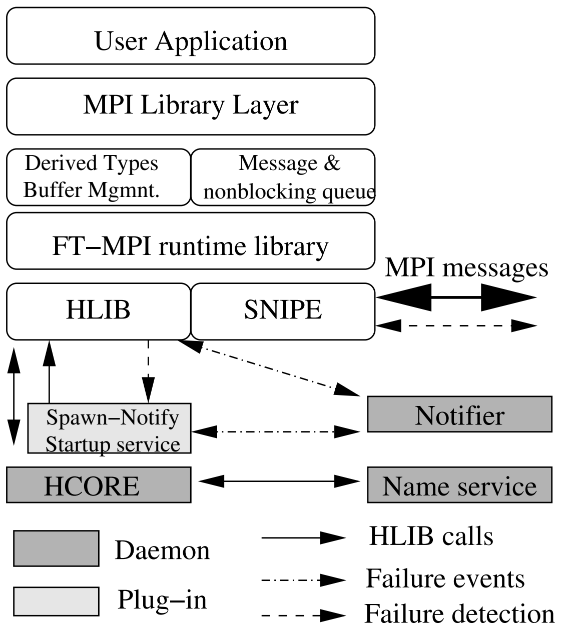

Figure 1. Architecture of HARNESS and FT-MPI

It is important to note that the FTRTL shown in figure 1 gets 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:

- 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.
- 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(3log$_2$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 [8] version 7. All tests were performed on a PC-cluster consisting of 16 nodes, each having two 2.4 GHz Pentium IV processors. A Gigabit Ethernet network connects the nodes.

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.

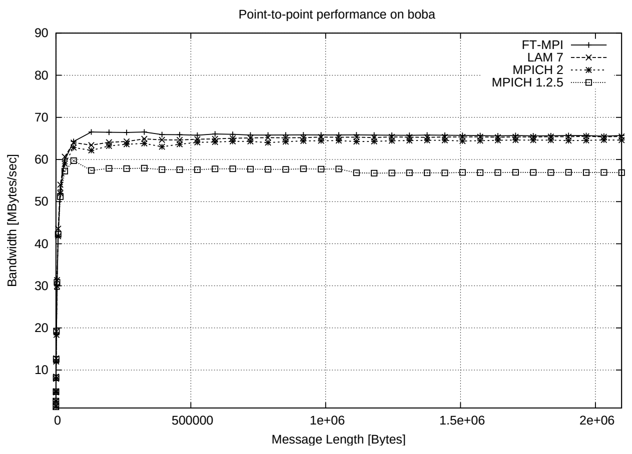

Figure 2. Achieved bandwidth with FT-MPI, LAM 7, MPICH 1.2.5 and MPICH 2

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, it is important, that FT-MPI achieves a comparable result with all current state-of-the-art MPI libraries for regular applications. 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 results achieved to the results with MPICH 1.2.5 and MPICH 2. LAM 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 not supported by LAM 7.

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 4 processes using 4 nodes on the same PC-cluster described in the previous section.

Problem	FT-MPI	MPICH 1.2.5	MPICH 2
t42.l17.240	93.6 sec	104.3 sec	93.9 sec
t85.l17.24	84.6 sec	96.2 sec	84.5 sec
t170.l3.12	63.7 sec	70.1 sec	64.0 sec

Table 1. Execution time of various problems with FT-MPI, MPICH 1.2.5 and MPICH 2

Table 1 presents the results achieved for these three test-cases. Generally speaking, FT-MPI and MPICH 2 are usually equally fast, although in most test cases, FT-MPI was slightly faster then MPICH 2. MPICH 1.2.5 is significantly slower than the other two MPI libraries for these test-cases. These results indicate, that FT-MPI is achieving top-end performance for non fault-tolerant applications and is therefore a real alternative to the other libraries.

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. In both cases, we assume, that the communicator mode used is REBUILD, which means that faulty processes are re-spawned by FT-MPI.

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 when a worker process dies, the master redistributes the work currently assigned to this process.

The first major question developing a fault tolerant application is, how can a process determine, whether it is a replacement for another, recently died process, or whether it belongs to the initial set of processes. FT-MPI offers two ways to determine this information. The first possibility involves the checking of the return value of MPI_Init, which is returning MPI_INIT_RESTARTED in case a processes is a replacement for another processes. This method is easy, it involves however an additional constant, which is not part of the MPI-1.2 specification. 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 completely portable and can also be used with any other non fault-tolerant MPI library.

As mentioned in section 2.1.3, there are two methods, how the application can detect an error: either the return code of every MPI function is checked, or the application registers right after MPI_Init an error-handler to his working communicator. The method how the recovery algorithm is invoked, also has influence on the state model required on both master and slave processes. In case the return code is checked after each MPI function, the slave process does not need any state model, since it can 'insist' on finishing a communication step properly, e.g.:

```
while ( 1 ) {
  rc = MPI_Recv ( .... );
  if ( rc == MPI_ERR_OTHER )
    recover();
  else
    break;
}
```

The master process has to maintain for each process its current state and which part of the work has been assigned to it. Each process can have one of the following states:

- AVAILABLE: this process is alive and no work is currently assigned to it
- WORKING: this process is alive and has some work assigned to it
- FINISHED: all work has been distributed, and the message indicating that everybody should exit has already been sent to this process
- FAULT: this process has died, its work will need to be redistributed.

Under normal conditions, the state of each process is changing from AVAILABLE to WORKING and back. In case an error occurs, the status of the process is changed to FAULT, until the process is re-spawned. In the case where the user would use the communicator mode BLANK instead of REBUILD, the state of the process would simply not be reset by the master.

The recovery algorithm routine contains first the re-instantiation of MPI_COMM_WORLD, which is done by calling MPI_Comm_dup. Next, all processes are calling the same sequence of collective operations for determining how many processes have died and who has died. Since the re-spawned processes are calling the same sequence after MPI_Init, correct MPI semantics are conserved by having all processes of MPI_COMM_WORLD participating in these collective operations. Finally, the status of the recovered process is set to AVAILABLE, while its previously assigned work is re-scheduled.

In case the user wants to avoid introducing an if-statement after every MPI routine, the application can register an error handler with its working communicator. The error handler has to call the same sequence of routines as mentioned in the previous paragraph. However, it has additional effects on the code as well. As an example, the master still has to know, whether the MPI_Recv operation from a certain process has succeeded or not (e.g. when collecting results from each process), before calculating some global results. Therefore, we introduced an additional state on the master node indicating, whether the receive operation has succeeded called RECEIVED. Thus, a worker processes has now to change its state from AVAILABLE to WORKING to RECEIVED before reaching AVAILABLE again. Furthermore, the master has to ensure, that certain states can just be reached from certain other changes. As an example, a process can reach the state AVAILABLE from the states RECEIVED and FAULT, while the state RECEIVED can never be reached directly from the state AVAILABLE. In our example, the `mark_*` routines guarantee, that just valid state transitions are executed. The pseudo-code for the master using error handlers appear as shown below.

```
/* 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],....);
         mark_working(proc);
      }

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

 /* Perform global calculation */
 for ( proc=1; proc<maxproc; proc++)
    if ( state[proc] == RECEIVED ) {
       workperformed += workid[proc];
       mark_available(proc);
    }

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

5.2 A parallel, fault-tolerant CG-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 conjugate gradient equation solver (PCG) in a fault tolerant manner. The parallel application has be 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 an MPI function returns MPI_ERR_OTHER indicating that an error has occurred, all processes *jump* to the recovery section in the code, perform the necessary recovery 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 matrix using the Boeing/Harwell format can be used for simulations. Table 2 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 (which is one less than the overall number of processes used in the simulation). The third column contains the execution time required to achieve a solution with the required precision. Finally, the fifth column is showing the recovery time for the used number of processes, in case a processes dies.

Problem size	No. of procs.	Exec. time [sec]	Recovery time/ratio [sec]/[%]
4,054	4	5	16/320
428,650	8	189	34/18
2,677,324	16	1162	69/6

Table 2. Execution time of various problem sizes with FT-MPI, and the recovery time for the according number of processes

Like table 2 indicates, recovering a from an exit-event of a processes takes between 16 seconds for 4 processes to 69 seconds for 16 processes. Most of this time is spent in a multi-phase commit protocol between the processes, since FT-MPI is capable of recovering from death events during another recovery procedure. Clearly, the recovery time is also the setting the limits, when recovering from an error makes sense, and when it does not. For example, for the four processes case shown in table 2, the recovery time is exceeding the overall execution time of the simulation by a factor of 3, and probably error recovery in this case is questionable. The next two cases however show, that for large and long running applications, the recovery time is less relevant, while the user still receives the result of his simulation in approximately the same time, even if an node failure is occurring during the simulation.

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 the extended functionality of failure-recovery. FT-MPI is however not an automatic checkpoint/recovery system, but 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 user application. Defining and implementing a consistent state in the application is however the responsibility of the end-users and 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. Thus, the overhead due to the fault-tolerant features of the libraries are small, and make FT-MPI an appealing alternative.

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 number 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.

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$^{\mathrm{TM}}$: 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 *Super-Computing*, 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.