A Fault-Tolerant Communication Library for Grid Environments

Edgar Gabriel, Graham E Fagg, Antonin Bukovsky, Thara Angskun,
and Jack J Dongarra

Innovative Computing Laboratory,
Department of Computer Science, Suite 413, 1122 Volunteer Blvd.,
University of Tennessee, Knoxville, TN-37996-3450, USA.
and
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN

{egabriel, fagg, tone, angskun, dongarra}@cs.utk.edu

Abstract

With increasing numbers of processors and applications running in virtual Grid environments, application level fault-tolerance is getting more of an important issue. 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 some tools supporting end-users during the application development step with FT-MPI are presented. Furthermore, a performance comparison of FT-MPI to the most relevant MPI-libraries for point-to-point benchmarks and the High Performance Linpack Benchmark, is shown.

1. Introduction

With increasing quality of the local, national and international networks, users have access higher quantities and to more powerful machines than ever in human history. While accessing and using these computing systems is straightforward for smaller number of machines, it becomes for larger numbers a science of its own: the Grid. The Grid is generally seen as a concept for coordinated resource sharing in virtual organizations, combining the resources of several institutions based on the demands of the application and just for the lifetime of the application.

Applications running on several machines can use various methods for data exchange between the processes. File transfer (e.g. FTP, SCP, GridFTP[20]), socket-based communication or RPC-like systems are just some examples. For scientific applications, the MPI[5][8] specification is meanwhile a widespread de-facto standard. Therefore it is not surprising, that several MPI-implementations optimized for Grid-environments are currently available (MPICH-G2[15], PACX-MPI[16], Stampi[17], MPI_Connect[18], and MagPIe[19]).

One problem that none of these implementations handle, is how to proceed when one or several processes become unavailable during runtime. This might happen because a machine or a node has crashed or is not reachable or because of networking problems. In fact, with increasing numbers of processors on both, single machines and in distributed Grid-environments, increasing numbers of applications will have to face node-failures. MPI in its current specification gives the user the choice between two possibilities how to handle a failure. The first one, the default mode, 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 for each aborted MPI application possibly large numbers of CPU hours wasted and lost, but also for very long running and security relevant applications this might not be an option at all. The relevance of this problem can also be seen by the numerous efforts in this area , e.g. FT-MPI[13], MPI/FT[21], MPI-FT[6], MPICH-V[11], LA-MPI[22].

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 do a detailed comparison of FT-MPI to the most relevant, most recent, non fault-tolerant MPI implementations. The structure of the paper is like follows. In section 2 we compare FT-MPI to related project in the area of fault-tolerant MPI libraries. In section 3 we present then the semantics, the concept and some tools for FT-MPI. Section 4 focuses on the performance comparison of FT-MPI with other MPI libraries for point-to-point benchmarks as well as for complex applications. Section 5 finally presents the current status of FT-MPI and presents the ongoing work in this area.

2. 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 [2] 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 [3]. Unlike Co-Check MPI, Starfish MPI uses its own distributed system to provide built in check-pointing.

MPICH-V[11] from Universit´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[22] is a fault-tolerant version of MPI from the Los Alamos National Laboratories. 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[21] 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 [6] 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.

3. 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.

3.1. 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.

3.1.1 Communicator and communication handling

Once a communicator has an error state it can only recover by rebuilding it, using a modified version of one of the MPI communicator build functions such as MPI_Comm_{create, split or dup}. Under these functions 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.

3.1.2 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.

3.1.3 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.

3.2 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[1]. One of the aims of HARNESS was to provide a framework for distributed computing much like PVM[29] 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.

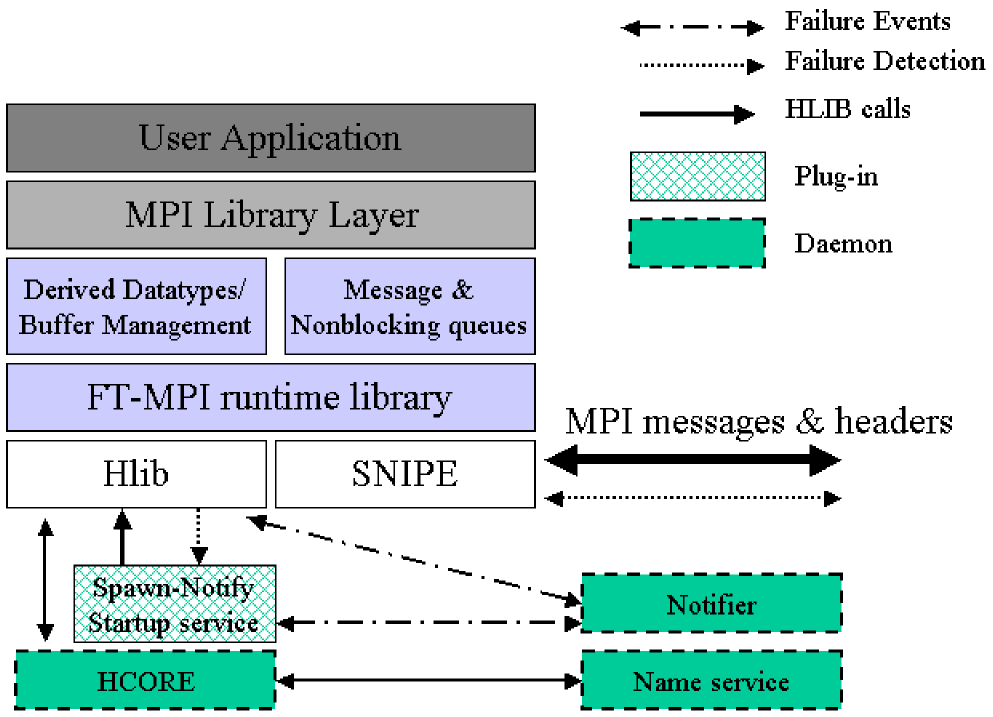


Figure 1. FT-MPI Architecture

3.2.1 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/marshalling (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 [13]. FT-MPI also implements a number of tuned MPI collective routines, which are further discussed in [14]. 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[4] 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 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.

3.2.2 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.

3.2.3 G_HCORE services for FT-MPI

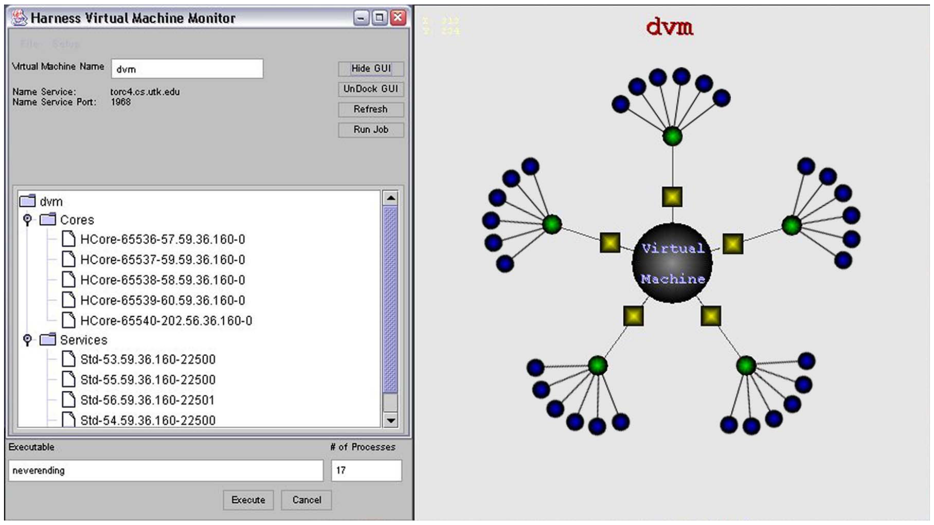
Services required by FT-MPI break down into two main categories:

- 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.

3.3 Tools for FT-MPI

Several tools developed simultaneously with FT-MPI for both fault-tolerant and non fault-tolerant usage are supporting FT-MPI. First, a *console* has been developed which provides a command line interface for adding or removing a host from the current virtual machine. Providing the similar commands to its PVM counterpart, the user is able to determine both runtime information from the VM, in addition to how many application processes are currently executing. The console is therefore an essential tool especially for large configurations. A graphical front-end to the console implemented in Java further enhances the user-friendliness of the HARNESS and FT-MPI environment (see figure 2).



```
Virtual Machine Name   dvm
Name Service:          torc4.cs.utk.edu
Name Service Port:     1968

dvm
  Cores
    HCore-65536-57.59.36.160-0
    HCore-65537-59.59.36.160-0
    HCore-65538-58.59.36.160-0
    HCore-65539-60.59.36.160-0
    HCore-65540-202.56.36.160-0
  Services
    Std-53.59.36.160-22500
    Std-55.59.36.160-22500
    Std-56.59.36.160-22501
    Std-54.59.36.160-22500

Executable    neverending
# of Processes    17
```

Figure 2. Screenshot of the Java-GUI for the console showing nameservice, startup-daemons and application processes

Second, the tracing library MPE [27] has been ported to work with FT-MPI. Using MPE, the user can generate tracefiles of its applications running with FT-MPI, which than can be analyzed using a visualizer such as jumpshot [26]. Performance analysis tools like jumpshot are for the development and tuning of parallel applications an important building block, since they guide the user to performance problems and bottlenecks in his parallel application. The user can however use MPE and jumpshot in a transparent manner only if his application is terminating correctly, since the tracefile is only written at the finalize step.

4. Performance comparison to non fault-tolerant MPI libraries

In this section we would like to compare the performance of FT-MPI to the performance achieved with the the most widely used, non fault-tolerant MPI implementations. These are MPICH [25](version 1.2.5) and LAM[28] (version 6.5.9). Additionally, we are also using the beta-version of the new MPICH2 library (version 0.9.3). All tests were performed on a PC-cluster consisting of 16 nodes, each node having two 2.4 GHz Pentium IV processors. A Gigabit Ethernet network connects the nodes.

4.1 Point-to-point performance tests

The first set of benchmarks were using the mpptest – testsuite [24], available with the MPICH distribution. We conducted three different tests, a short-message test (0-1024 bytes), a test for medium size messages (0-128Kbytes) and a long message test (0-2Mbytes) for determining the maximum achievable bandwidth. Figure 3 presents the results for the short message test. The fastest library for short-messages was LAM, followed by MPICH2. FT-MPI is between MPICH2 and MPICH 1.2.5 for small messages.

For medium size messages, the ranking of the libraries is changing at around 20 Kbyte messages. LAM and MPICH1.2.5 become from this point on around 10% slower than MPICH2 and FTMPI. As a consequence, MPICH2 and FT-MPI show the best performance also for long-messages, achieving a bandwidth of around

67 Mbytes/second on the Gigabit Ethernet connection (FT-MPI), respectively 65 Mbytes/second (MPICH2).

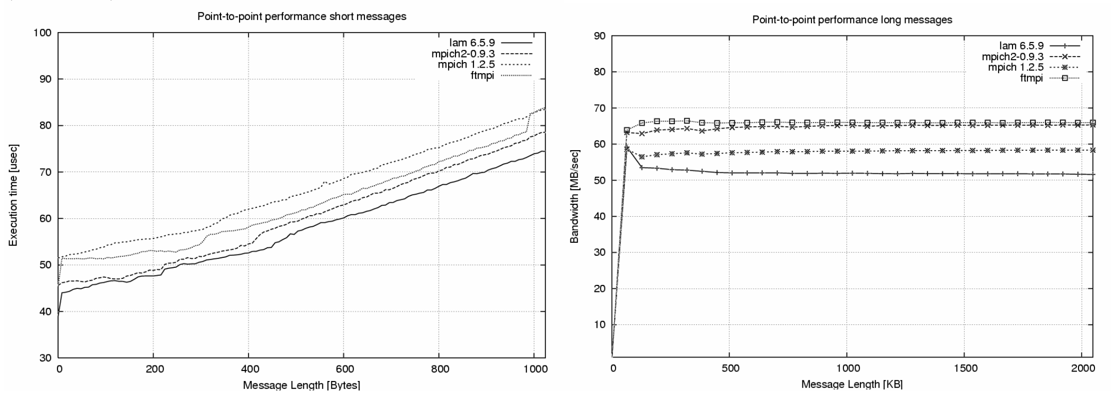


Figure 3: Performance results for all MPI libraries for short-messages (left) and long messages (right).

4.2 Results for High Performance Linpack (HPL)

In this section we would like to present the performance results achieved with the High Performance Linpack Benchmark [7] as a representative of a complex end-user application. From the many optimization possibilities and performance parameters, which can be used to tune HPL, we chose a single test case and modified a single parameter, the block-size. Changing the block-size in HPL has not only an effect on the data distribution, but also on the message sizes sent. Therefore, by modifying this parameter, we can show, which MPI-library is achieving the best results for different block-sizes/message lengths.

The tests were conducted for a problem size of 6000, using 4 processes. Table 1 shows the results of our measurements. Up to the block-size of 80, FT-MPI is second in all tests, just MPICH2 is achieving better performance than FT-MPI. For block-sizes larger than 80, FT-MPI and MPICH1.2.5 are nearly equal. The overhead of FT-MPI compared to the according fastest library is between 2% and 5%, which is moderate taking into account, that FT-MPI is offering additional functionality compared to the other libraries. In the following, we would like to focus on three testcases, using block-sizes of 48, 80 and 240.

Blocksize	**FT-MPI**	**MPICH 1.2.5**	**LAM 6.5.9**	**MPICH 2 - 0.9.3**
16	25.25 sec	28.84 sec	25.78 sec	24.97 sec
32	20.51 sec	20.61 sec	20.78 sec	19.90 sec
48	28.00 sec	28.16 sec	28.31 sec	27.43 sec
80	18.18 sec	18.18 sec	18.50 sec	17.41 sec
128	22.96 sec	22.86 sec	23.03 sec	22.07 sec
160	18.97 sec	18.89 sec	19.24 sec	18.03 sec
240	20.05 sec	19.88 sec	20.28 sec	19.09 sec

Table 1. Execution time of HPL with different bocksizes and MPI libraries

There are two effects which are relevant for the communication performance of HPL. On one hand, there is a large number of small messages (0-4KB), which is independent of the block-size. The most relevant short messages are occurring 6000 times per test. On the other hand, we have increasing message-sizes and decreasing number of messages for increasing block-sizes, as shown in Table 2. While the overall amount of data remains roughly constant, the number of messages decreases with increasing block-size, and the messages get longer. Thus, while FT-MPI shows a good performance for long messages (as shown in section 4.1), it still suffers from a higher latency than MPICH-2. This is probably the reason for the higher execution time of FT-MPI. LAM is slower compared to FT-MPI, probably mainly because of its limited performance for large messages. The performance of FT-MPI and MPICH1.2.5 are comparable for large block sizes, however, FT-MPI has an advantage for smaller block-sizes.

Blocksize	No. of msg. >20 KB	No. of msg. >100 KB	No. of. msg. > 1 MB
48	142	811	27
80	53	476	73
240	12	57	132

Table 2. Number of large messages for different block-sizes.

5. Conclusions

FT-MPI is an attempt to provide application programmers with different methods of dealing with failures within MPI application than just check-point and restart. It is hoped that by experimenting with FT-MPI, new applications methodologies and algorithms will be developed to allow for both high performance and the survivability required by both unreliable GRIDs and the next generation of terra-flop and beyond machines. FT-MPI in itself is already proving to be a useful vehicle for experimenting with self-tuning collective communications, distributed control algorithms, various dynamic library download methods and improved sparse data handling subsystems, as well as being the default MPI implementation for the HARNESS project.

FT-MPI implements currently the full MPI 1.2 specification as well as several sections of the MPI-2 document. Furthermore, FT-MPI has full tool-support for both normal MPI features such as profiling and debugging via MPE, as well as additional support for fault-tolerance. The first major release will be available end second quarter 2003. Future work in the FT-MPI library system will concentrate on developing a number of drop-in library templates or skeletons to simplify the construction of fault tolerant applications.

6. References

1. Beck, Dongarra, Fagg, Geist, Gray, Kohl, Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. Scott, V. Sunderam, "HARNESS: a next generation distributed virtual machine", Journal of Future Generation Computer Systems, (15), Elsevier Science B.V., 1999.
2. G. Stellner, “CoCheck: Checkpointing and Process Migration for MPI”, In Proceedings of the International Parallel Processing Symposium, pp 526-531, Honolulu, April 1996.
3. Adnan Agbaria and Roy Friedman, “Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations”, In the 8th IEEE International Symposium on High Performance Distributed Computing, 1999.
4. Graham E. Fagg, Keith Moore, Jack J. Dongarra, "Scalable networked information processing environment (SNIPE)", Journal of Future Generation Computer Systems, (15), pp. 571-582, Elsevier Science B.V., 1999.
5. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. MPI- The Complete Reference. Volume 1, The MPI Core, second edition (1998).
6. Soulla Louca, Neophytos Neophytou, Adrianos Lachanas, Paraskevas Evripidou, “MPI-FT: A portable fault tolerance scheme for MPI”, Proc. of PDPTA ’98 International Conference, Las Vegas, Nevada 1998.
7. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. Scalapack: A linear algebra library for message-passing computers. In Proceedings of 1997 SIAM Conference on Parallel Processing, May 1997.
8. William Gropp, Ewing Lusk, and Rajeev Thakur , “Using MPI-2: Advanced Features of the Message Passing Interface”, MIT Press, 1st Edition, Feburary 2000.
9. F. Berman, A. Chen, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellow-Crummey, D. Reed, L. Torczon, and R.Wolski, “The GrADS Project”, International Journal of High Performance Computing Applications, Vol 15(4), pp. 327-344, Sage Science Press, Winter 2001.
10. I. Foster and C. Kesselmann, *The GRID: Blueprint for a new computing infrastructure*, Morgan Kaufmann, San Francisco, 1999.

11. George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, Anton Selikhov, “MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes”, *In Proceedings of SuperComputing 2002. IEEE, Nov.*,2002.
12. Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith, *PETSc 2.0 Users Manual* Argonne National Laboratory, ANL-95/11 - Revision 2.0.29, 2000.
13. Graham Fagg, Antonin Bukovsky, and Jack Dongarra, HARNESS and Fault Tolerant MPI, Parallel Computing, Volume 27, Number 11, pp 1479-1496, October 2001, ISSN 0167-8191
14. Sathish S. Vadhiyar, Graham E. Fagg, and Jack J. Dongarra, Performance Modeling for Self Adapting Collective Communications for MPI, LACSI Symposium 2001, October 15-18, Eldorado Hotel, Santa Fe,NM.
15. N. Karonis, B. Toonen, and I. Foster, MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface, Journal of Parallel and Distributed Computing, to appear 2003.
16. Edgar Gabriel, Michael Resch, and Roland Ruehle, Implementing MPI with Optimized Algorithms for Metacomputing, in Anthony Skjellum, Purushotham V. Bangalore, Yoginder S. Dandass, ‘Proceedings of the Third MPI Developer’s and User’s Conference’, MPI Software Technology Press, Starkville, Mississippi, 1999.
17. T. Imamura, Y. Tsujita, H. Koide, H. Takemiya, An Architecture of Stampi: MPI library on a cluster of parallel computers, in J. Dongarra, P. Kacsuk, N. Podhorszki (Eds.) ‘ Recent Advances in Parallel Virtual Machine and Message Passing Interface’, Lecture Notes in Computer Science vol. 1908, pp. 200-207, Springer, Berlin 2000.
18. G.E. Fagg, K. S. London, J. J. Dongarra, MPI_Connect: Managing heterogeneous MPI application interoperation and process control, in V. Alexandrov, J. Dongarra (Eds.), ‘Recent Advances in Parallel Virtual Machine and Message Passing Interface’, Lecture Notes in Computer Science vol. 1947, pp. 93-96, Springer, Berlin, 1998.
19. T. Kielmann, R.F.H. Hofman, H.E. Bal, MagPIe: MPI’s collective communication operations for clustered wide area systems, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP’99), pp.131-140, ACM, 1999.
20. W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, L. Liming, S. Meder, S.Tuecke, GridFTP Protocol Specification, GGF GridFTO Working Group Document, September 2002.
21. R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjellum, Y. Dandass, and M. Apte, MPI/FT: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing, in Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid, held in Melbourne, Australia, 2001.
22. Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger and Mitchel W. Sukalski, A Network-Failure-Tolerant Message-Passing System For Terascale Clusters, ICS02, June 22-26, 2002, New York, New York, USA.
23. William Gropp and Ewing Lusk, Fault Tolerance in MPI Programs, to appear in Journal of High Performance Computing and Applications, 2003.
24. William Gropp and Ewing Lusk, Reproducible measurements of MPI performance characteristics, in Jack Dongarra, Emilio Luque, Tom Margalef (Eds.), ‘Recent Advances in Parallel and Virtual Machine and Message Passing Interface’, Lecture Notes in Computer Science vol. 1697, Springer, Berlin 1999.
25. W. Gropp, E. Lusk, N. Doss, and A. Skjellum, A high-performance, portable implementation of the MPI message passing interface standard, Parallel Computing, 22(6):789-828, September 1996.
26. Performance Visualization for Parallel Programs, World Wide Web: http://www-unix.mcs.anl.gov/perfvis/software/
27. User’s Guide for MPE: Extentions for MPI Programs, World Wide Web : http://www-unix.mcs.anl.gov/mpi/mpich/docs/mpeman/mpeman.html
28. G. Burns, R. Daoud, Robust MPI Message Delivery Through Guaranteed Resources, MPI Developers Conference, University of Notre Dame, June 1995.
29. G. Geist, J. Kohl, R. Manchel, and P. Papadopolous, New Features of PVM 3.4 and Beyond, PVM Euro User’s Group Meeting, pp. 1-10, September, 1995.