Building and using an Fault Tolerant MPI implementation

Graham E Fagg [+*]
Jack J Dongarra[*]

[+]High Performance Computing Center Stuttgart
Allmandring 30, D-70550 Stuttgart, Germany.

[*]Department of Computer Science, Suite 413, 1122 Volunteer Blvd.,
University of Tennessee, Knoxville, TN-37996-3450, USA.

fagg@hlrs.de
dongarra@cs.utk.edu

Abstract

In this paper we discuss the design and use of a fault tolerant MPI (FT-MPI) that handles process failures in a way beyond that of the original MPI static process model. FT-MPI allows the semantics and associated modes of failures to be explicitly controlled by an application via a modified functionality within the standard MPI 1.2 API. Given is an overview of the FT-MPI semantics, architecture design, example usage and sample applications. A short discussion is given on the consequences of designing a fault tolerant MPI both in terms of how such an implementation handles failures at multiple levels internally as well as how existing applications can use new features while still remaining within the MPI standard.

1. Introduction

MPI [6] is the current the standard message passing system used to build high performance applications for both clusters and dedicated MPP systems. Initially MPI was designed to allow for very high efficiency and thus performance on a number of early 1990s MPPs, that at the time had limited OS runtime support. This led to the current MPI design of a static process model. While this model was possible to implement for MPP vendors, easy to program for, and more importantly something that could be agreed upon by a standards committee. The second version of MPI standard known as MPI-2 [10] did include some support for dynamic process control, although this was limited to the creation of new MPI process groups with separate communicators. These new processes could not be merged with previously existing communicators to form intra-communicators needed for a seamless single application model and were limited to a special set of extended collectives (group) communications.

The MPI static process model suffices for small numbers of distributed nodes within the currently emerging masses of clusters and several hundred nodes of dedicated MPPs. Beyond these sizes the mean time between failure (MTBF) of CPU nodes becomes a factor. As attempts to build the next generation Peta-flop systems advance, this situation will only become more adverse as individual node reliability becomes out weighted by orders of magnitude increase in node numbers and hence node failures. Current GRID [16] technologies such as GLOBUS [13] also provide for middleware services such as naming, resource discovery that are robust and handle expected failures gracefully. Unfortunately the MPI message passing library for Globus, MPICH-G [14] is not expected to handle loss of MPI processes or partitioning of networks gracefully and failures still lead to pathological failure of applications unless special precautions are taken such application check-pointed discussed further in the next section.

The aim of FT-MPI is to build a fault tolerant MPI implementation that can survive failures, while offering the application developer a range of recovery options other than just returning to some previous check-

pointed state. FT-MPI is built on the HARNESS [1] meta-computing system, and is meant to be used as the HARNESS default application level message passing interface. Its design allows it to easily ported to other GRID environments by porting of its modular services that are implemented in the form of short lived daemons.

2. Check-point and roll back versus 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. In this implementation, all processes would flush their messages queues to avoid in flight messages getting lost, and then they would all synchronously check-point. At some later stage if either an error occurred or a task was forced to migrate to assist load balancing, the entire MPI application would be rolled back to the last complete check-point and be restarted. This systems main drawback being the need for the entire application having to check-point synchronously, which depending on the application and its size could become expensive in terms of time (with potential scaling problems). A secondary consideration was that they had to implement a new version of MPI known as tuMPI as updating MPICH was considered too difficult.

Another system that also uses check-pointing but at a much lower level is StarFish MPI [3]. Unlike Co-Check MPI which relies on Condor, Starfish MPI uses its own distributed system to provide built in check-pointing. The main difference with Co-Check MPI is how it handles communication and state changes which are managed by StarFish using strict atomic group communication protocols built upon the Ensemble system, and thus avoids the message flush protocol of Co-Check. Being a more recent project StarFish supports faster networking interfaces than tuMPI.

The project closest to FT-MPI known to the author is the Implicit Fault Tolerance MPI project MPI-FT [7] 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. This system is not a full checkpoint system in that it assumes any data (or state) can be rebuilt using just the knowledge of any passed messages, which might not be the case for non deterministic unstable solvers.

MPICH-V[17] 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. Ping-pong performance of MPICH-V compared to MPICH-p4 is around 50%, although application performance is usually much better. In the case of the NAS BP benchmark the overhead for MPICH-V compared to MPICH over P4 varies between 6% and 20%. Handling of a failure is automatic and transparent to the user, although currently only master-slave or SPMD applications are supported.

FT-MPI has much lower overheads compared to the above check-pointing 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.

Currently GRID application efforts such as GrADS [11] primarily focus on gaining high performance from GRIDs rather than handling failures, although current efforts at the University of Tennessee [12] involve check-pointing distributed applications to improve fault tolerance. Unlike the above check-pointing systems that rely on local disks for check-pointed data storage, the current GRADS effort is experimenting with replicated distributed storage built on top of the IBP [15] system to improve both availability and performance. This system is also a user-level check-pointing scheme rather than process level and thus would benefit from avoiding rescheduling as provided by FT-MPI.

3. 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 (and it is unclear if we can even *free* 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. Some details of failure detection is given in 4.1.

The typical MPI semantics is from OK to Failed which then causes an application abort. By allowing the communicator to be in an intermediate state we allow the application the ability to decide how to alter the communicator and its state as well as how communication within the intermediate state behaves.

3.1. Failure modes

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. Note, this might not be all current communicators as we support MPI-2 dynamic tasks and thus multiple MPI_COMM_WORLDS.

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.

3.2. 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.

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 is available though the cached attributes interface in MPI as discussed in section 3.4 below.

3.3. 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.

3.4. FT-MPI notification of failures

The MPI standard does not indicate how errors are reported beyond standard return codes and error classes to provide additional information. Without altering the meaning of the standard, FT-MPI utilizes these mechanisms so that applications that have been adapted to FT-MPI still compile and link correctly on other MPI implementations.

To remain within the standard FT-MPI notifies the application with a single return code MPI_ERR_OTHER that an error has occurred and then makes additional information available via the attribute caching mechanism. A human readable form of the failure is also provided via a MPI error class using the MPI error string function.

Two forms of essentially the same information are made available to the application. The first form returns the error information for a complete communicator in terms of the number of failures per rank since the last recovery, The second form returns the failed ranks in the order that they were detected *locally*. This ordering is only consistently globally in terms of the total failures not the ordering reported at each node unless the FTMPI_NOTIFIER daemon is used to force ordering of events.

```
/* pre-defined key value */
key = FT_MPI_LIST_NUM_FAILED;  /* key for finding number of failure events */
key2 = FT_MPI_LIST_FAILED;         /* key for getting pointer to failures in a list */

rc= MPI_func (comm…)
If (rc==MPI_ERR_OTHER) {
       rc = MPI_Comm_get_attr (comm, key, &num_failed, &flag);
       rc = MPI_Comm_get_attr (comm, key2, &failed_ptr, &flag);
       for (i=0;i<num_failed;i++)
              printf("failure %d was rank %d\n", i+1, failed_ptr[i]);
       }
```

Example 1. Checking for order of failures

```
key = FT_MPI_COM_NUM_FAILED;  /* key for finding how many individual ranks failed */
key2 = FT_MPI_COM_FAILED;          /* key for accessing complete failure map of a communicator */

rc= MPI_Send (----, com);
If (rc==MPI_ERR_OTHER) {
       rc = MPI_Comm_get_attr (comm, key, &num_failed, &flag);
       rc = MPI_Comm_get_attr (comm, key2, &failed_ptr, &flag);
       /* check list of failures */
       failed_how_many_times = failed_ptr [rank];
       }
```

Example 2. Accessing failures via process RANK

3.5. FT-MPI basic usage

Simple usage of FT-MPI would be in the form of an error check and then some corrective action such as a communicator rebuild. A typical code fragment is shown below in example 3, where on an error the communicator is simply rebuilt and reused:

```
rc= MPI_Send (----, com);
If (rc==MPI_ERR_OTHER) {
       MPI_Comm_dup (com, newcom);      /* collective recovery occurs here! */
       MPI_Comm_free (com);
       com = newcom;
       }
/* continue.. */
```

Example 3. Simple FT-MPI send usage

Some types of computation such as SPMD master-worker codes only need the error checking in the master code if the user is willing to accept the master as the only point of failure. Example 4 below shows how complex a master code can become. In this example the communicator mode is BLANK and communications mode is CONT. The master keeps track of work allocated, and on an error just reallocates the work to any 'free' surviving processes. Note, the code has to check to see if there are any surviving workers remaining after each death is detected.

```
rc = MPI_Bcast ( initial_work…);
```

```
if(rc==MPI_ERR_OTHER)reclaim_lost_work(…);
  while ( ! all_work_done) {
  if (work_allocated) {
    rc = MPI_Recv ( buf, ans_size, result_dt,
                    MPI_ANY_SOURCE,   MPI_ANY_TAG, comm, &status);
    if (rc==MPI_SUCCESS) {
                          handle_work (buf);
                          free_worker (status.MPI_SOURCE);
                          all_work_done--;
                          }
    else {
           reclaim_lost_work(status.MPI_SOURCE);
          if (no_surviving_workers) { /* ! do something ! */ }
         }
  } /* work allocated */
/* Get a new worker as we must have received a result or a death */
  rank=get_free_worker_and_allocate_work();
  if (rank) {
    rc = MPI_Send (… rank… );
    if (rc==MPI_OTHER_ERR) reclaim_lost_work (rank);
    if (no_surviving_workers) { /* ! do something ! */ }
  } /* if free worker */
} /* while work to do */
```

Example 4. FT-MPI Master-Worker code

3.6. FT-MPI usage within existing message passing libraries

Many real world parallel applications use numeric libraries such as SCALAPACK[8] and PETSc[18] which themselves use MPI internally through multiple layers. Altering such libraries by changing each occurrence of each MPI call is impractical and error prone.

A more elegant solution is to use the MPI error handling functions to automatically handle the errors for the application. When combined with the long jump mechanism in the C language this can provide a very simple solution to many classes of error handling. A typical program flow for an application is given in Figure 1. If the application already contains user level check-pointing then only the initial startup section of the code needs to be altered. The flow within a normal process would proceed as follows:

1. MPI_Init would indicate if the process was started normally via MPIRUN or was a restarted node within an application.
2. If the process was normal, then the application would install the MPI error handler that they wrote as shown below in code example 5.
3. The process would set a long jump so that it could return to the top level functions where it can correctly manage program flow during a recovery. This is required as a failure could be many levels of function calls later.
4. The code would call the numeric library containing MPI calls (i.e. a parallel solver)
5. If completed successfully the code would enter MPI_finalize and terminate normally.

During the execution if an error occurred, the FT-MPI runtime library would catch it and as soon as the program enters a MPI routine, flow control would be passed to the MPI error handler the user provided in 2 above. At this point the users application could block on a communicator create/duplicate function after which they would probably load the user level checkpoint data. After recovery they would then jump back to the top level of the application, reset the jump and then continue as per 3 above.

A restarted process would discover from the MPI_Init function that it was restarted and would then load any recovery data rather than initial data, install the error handler and continue as a normal process.

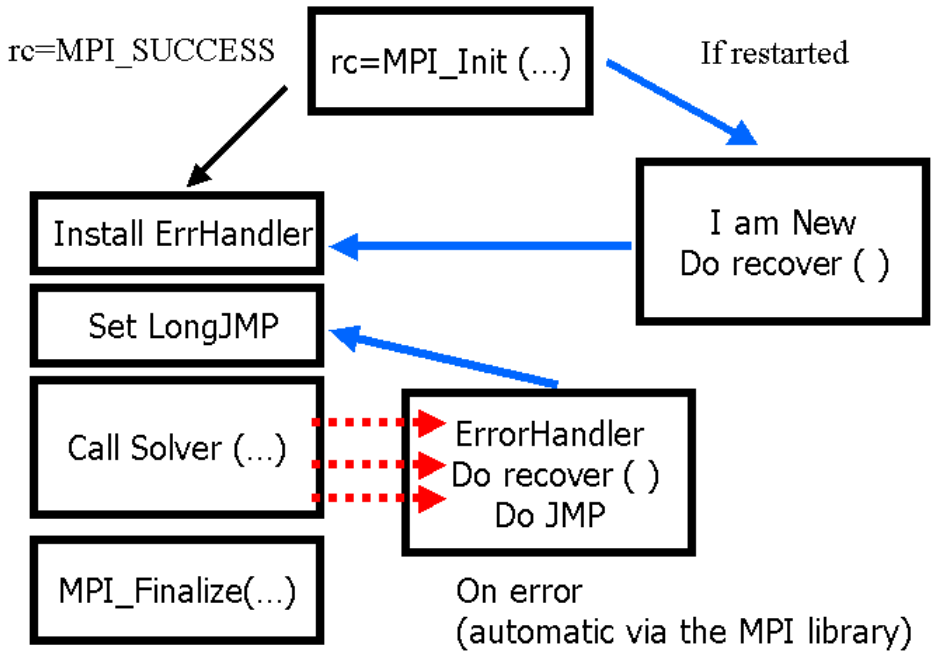

Figure 1. Flow control in a typical FT-MPI application using MPI Error Handlers.

```
ehf = (MPI_Handler_function *) (&errhandleruserfunc); /* get handle to my error handler */

MPI_Errhandler_create (ehf, &errh);          /* create MPI handle to my function */
MPI_Errhandler_get (MCW, &errh_org);         /* get original MPI handler */
MPI_Errhandler_free (&errh_org);
MPI_Errhandler_set (MCW, errh);              /* replace default with my function */
```

Example 5. Installing an error handler under MPI

4. FT_MPI Implementation details

FT-MPI is a partial MPI-2 implementation. It currently contains support for both C and Fortran interfaces, all the MPI-1.2 function calls required to run both the PSTSWM [5] and BLAS [9] applications. BLAS is supported so that SCALAPACK [8] applications can be tested. Currently only some the dynamic process control functions from MPI-2 are supported.

The current implementation is built as a number of layers as shown in figure 2. Operating system support is provided by either PVM or the C HARNESS *G_HCORE*. Although point to point communication is provided by a modified SNIPE_Lite communication library taken from the SNIPE project [4].

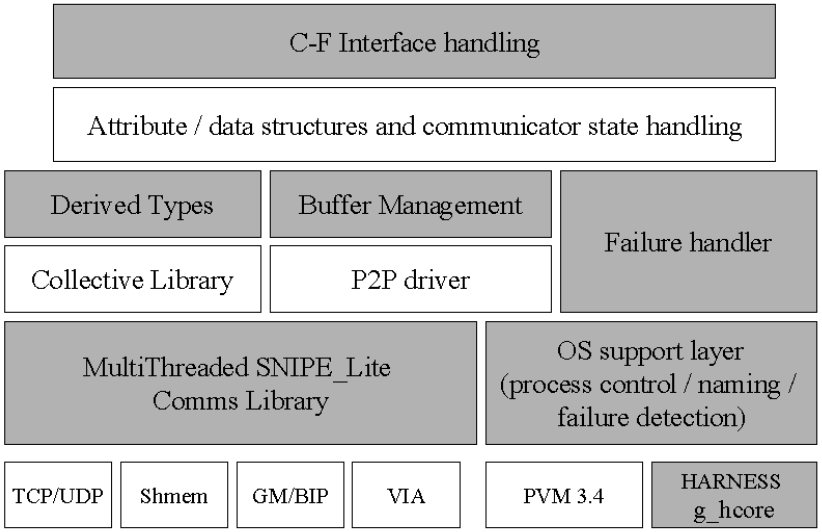

Figure 2. Overall structure of the FT-MPI implementation.

A number of components have been extensively optimized, these include, derived data types [19] and message buffers and collective communications[20].

4.1. Failure detection

It is important to note that the failure handler shown in figure 2, gets notification of failures from both the point to point communications libraries as well as the OS support 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 handler 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. An additional daemon know as the FTMPI_NOTIFER can be used to guarantee ordered delivery of failure notification messages and thus aid in complex debugging.

The failure handler within the FTMPI runtime library relies on the conservation of event messages from the underlying system to build a coherent system state during recovery. A consequence of this is that temporary bi-sectioning of the network between G_HCORE startup daemons can lead to some processes being marked as failed and thus the sum of living tasks and failure events will remain constant.

4.2. Low-level message handling

Many MPI message passing libraries employ multiple message delivery schemes which vary with message size to provide a balance between performance, unexpected message buffering memory requirements and blocking semantics. GM for example switches between eager (always send) and rendezvous modes as the message size increases.

FT-MPI uses eager for performance on all blocking sends and switches to a token based system for large non-blocking messages. As with the failure detection, the handling of communication during failures relies on a guaranteed delivery of flow control messages and failure events.

During a failure all processes flush communications with all existing communication contexts. They complete all pending operations involving a remote process, until either they have received a flow control

message indicating that the process is entering a global state rebuild or a failure event for that process is received. Thus the number of flow control stop messages and death events of open connections must match the number of pre-failure open connections. This allows all/any processes in an eager send to always complete as their target guarantees emptying the pipe before entering the global recovery state, thus avoiding any deadlocks.

5. FT-MPI Performance

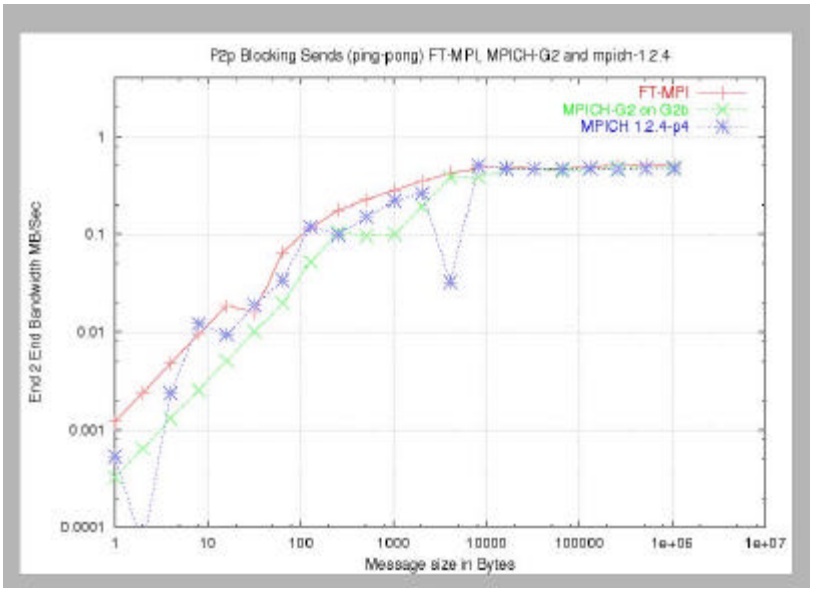

Figure 3. Point-to-point message performance of FT-MPI compared to various MPICH versions

Figure 3 shows the performance of FT-MPI for point-to-point messages compared to MPICH-p4 and MPICH-G2 under Globus 2.0. Further performance information can be obtained from [19-20]. As was stated in section 2, the performance of FT-MPI is not hindered by fault handling. Any additional costs of being fault tolerance is only occur at applications startup, during a failure recovery and during shutdown.

6. 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.

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.

7. 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. P. H. Worley, I. T. Foster, and B. Toonen, "Algorithm comparison and benchmarking using a parallel spectral transform shallow water model", Proccedings of the Sixth Workshop on Parallel Processing in Meteorology, eds. G.-R. Hoffmann and N. Kreitz, World Scientific, Singapore, pp. 277-289, 1995.
6. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. MPI- The Complete Reference. Volume 1, The MPI Core, second edition (1998).
7. 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.
8. 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.
9. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. Whaley, A Proposal for a Set of Parallel Basic Linear Algebra Subprograms, , *LAPACK Working Note #100*, CS-95-292, May 1995
10. William Gropp, Ewing Lusk, and Rajeev Thakur , "Using MPI-2: Advanced Features of the Message Passing Interface", MIT Press, 1st Edition, Feburary 2000.
11. 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.
12. A. Petitet, S. Blackford, J. Dongarra, B. Ellis, G. Fagg, K. Roche, and S. Vadhiyar, "Numerical Libraries and the Grid", International Journal of High Performance Computing Applications, Vol 15(4), pp. 359-374, Sage Science Press, Winter 2001.
13. I. Foster and C. Kesselmann, "The Globus Toolkit", in *The GRID: Blueprint for a new computing infrastructure*, edited by I. Foster and C. Kesselmann, pp. 259-278. Morgan Kaufmann, San Francisco, 1999.
14. I. Foster, and N. Karonis, "A Grid enabled MPI: Message passing in heterogeneous distributed computing systems", Proc. of SuperComputing 98 (SC98), Orlando, FL.
15. James S. Plank, Micah Beck, Wael R. Elwasif, Terry Moore, Martin Swany, Rich Wolski, "The Internet Backplane Protocol: Storage in the Network", NetStore99: The Network Storage Symposium, (Seattle, WA, 1999)
16. I. Foster and C. Kesselmann, *The GRID: Blueprint for a new computing infrastructure*, Morgan Kaufmann, San Francisco, 1999.
17. 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.
18. 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.
19. 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
20. 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.