Fault Tolerant MPI for the HARNESS MetaComputing system

Graham E Fagg, Antonin Bukovsky, Sathish Vadhiyar and Jack J Dongarra

Department of Computer Science, Suite 203, 1122 Volunteer Blvd.,
University of Tennessee, Knoxville, TN-37996-3450, USA.

fagg@cs.utk.edu

Abstract

Initial versions of MPI were designed to work efficiently on multi-processors which had very little job control and thus static process models. Subsequently forcing them to support a dynamic process model suitable for use on clusters or distributed systems would have reduced their performance. As current HPC collaborative applications increase in size and distribution the potential levels of node and network failures increase the need arises for new fault tolerant systems to be developed. Here we present a new implementation of MPI called FT-MPI that allows the semantics and associated modes of failures to be explicitly controlled by an application via a modified MPI API. Given is an overview of the FT-MPI semantics, design, example applications, debugging tools and some performance issues such as efficient group communications and complex data handling. Also discussed is the experimental HARNESS core (G_HCORE) implementation that FT-MPI is built to operate upon.

1. Introduction

Although MPI [11] is currently the de-facto standard system used to build high performance applications for both clusters and dedicated MPP systems, it is not without it problems. 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 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 starts becoming 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.

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 its default application level message passing interface.

2. Check-point and roll back verses 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 retro-fitting 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 [4], 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 by 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.

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 more detail on failure detection is given in 4.4.

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 effects 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:

- 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.
- CONT: All communication that is NOT to the effected/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.

3.3. Point to Point verses 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 all gather 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 usage

Typical 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, 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);
      com = newcom;     /* continue.. */
```

Some types of computation such as SPMD master-slave 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. The example 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 checks to see if there are surviving worker processes left 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 */
```

4. FT_MPI Implementation details

FT-MPI is a partial MPI-2 implementation in its own right. It currently contains support for both C and Fortran interfaces, all the MPI-1 function calls required to run both the PSTSWM [6] and BLACS applications. BLACS is supported so that SCALAPACK application 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 1. 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].

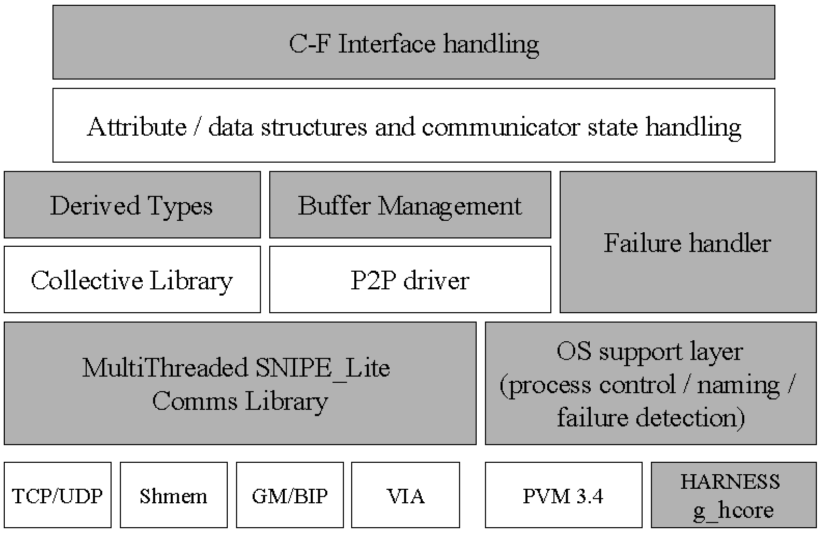

Figure 1. Overall structure of the FT-MPI implementation.

A number of components have been extensively optimized, these include:
- Derived data types and message buffers.
- Collective communications.
- Point to point communication using multi-threading.

4.1. Derived Data Type handling

MPI-1 introduced extensive facilities for user Derived DataType (DDT)[11] handling that allows for in effect strongly typed message passing. The handling of these possibly non-contiguous data types is very important in real applications, and is often a neglected area of communication library design [17]. Most communications libraries are designed for low latency and/or high bandwidth with contiguous blocks of data [14]. Although this means that they must avoid unnecessary memory copies, the efficient handling of recursive data structures is often left to simple iterations of a loop that packs a send/receive buffer.

4.1.1. FT-MPI DDT handling

Having gained experience with handling DDTs within a heterogeneous system from the PVMPI/MPI_Connect library [18] the authors of FT-MPI redesigned the handling of DDTs so that they would not just handle the recursive data-types flexibly but also take advantage of internal buffer management structure to gain better performance. In a typical system the DDT would be collected/gathered into a single buffer and then passed to the communications library, which may have to encode the data using XDR for example, and then segment the message into packets for transmission. These steps involving multiple memory copies across program modules (reducing cache effectiveness) and possibly precluding overlapping (concurrency) of operations.

The DDT system used by FT-MPI was designed to reduce memory copies while allowing for overlapping in the three stages of data handling:
- gather/scatter : Data is collected into or from recursively structured non-contiguous memory.
- encoding/decoding : Data passed between heterogeneous machine architectures than use different floating point representations need to be converted so that the data maintains the original meaning.
- send/receive packetizing : All of the send or receive cannot be completed in a single attempt and the data has to be sent in blocks. This is usually due to buffering constraints in the communications library/OS or even hardware flow control.

4.1.2. DDT methods and algorithms

Under FT-MPI data can be gathered/scattered by compressing the data type representation into a compacted format that can be efficiently transversed (not to be confused with compressing data discussed below). The algorithm used to compact data type representation would break down any recursive data type into an optimized maximum length new representation. FT-MPI checks for this optimization when the users application commits the data type using the MPI_Type_commit API call. This allows FT-MPI to optimize the data type representation before any communication is attempted that uses them.

When the DDT is being processed the actual user data itself can also be compacted into/from a contiguous buffer. Several options for this type of buffering are allowed that include:

- Zero padding: Compacting into the smallest buffer space
- Minimal padding: Compacting into smallest space but maintaining correct word alignment
- Re-ordering pack: Re-arranging the data so that all the integers are packed first, followed by floats etc. i.e. type by type.

The minimal and no padded methods are used when moving the data type within a homogeneous set of machines that require no numeric representation encoding or decoding. The zero padding method benefits slower networks, and alignment padded can in some cases assist memory copy operations, although its real benefit is when used with re-ordering.

The re-ordered compacting method shown in figure 2, is designed to be used when some additional form encoding/decoding takes place. In particular moving the re-ordered data, type by type through fixed XDR buffers improves its performance considerably. Two types of DDT encoding are supported, the first is the slower generic SUN XDR format and the second is simple byte swapping to convert between little and big endian numbers.

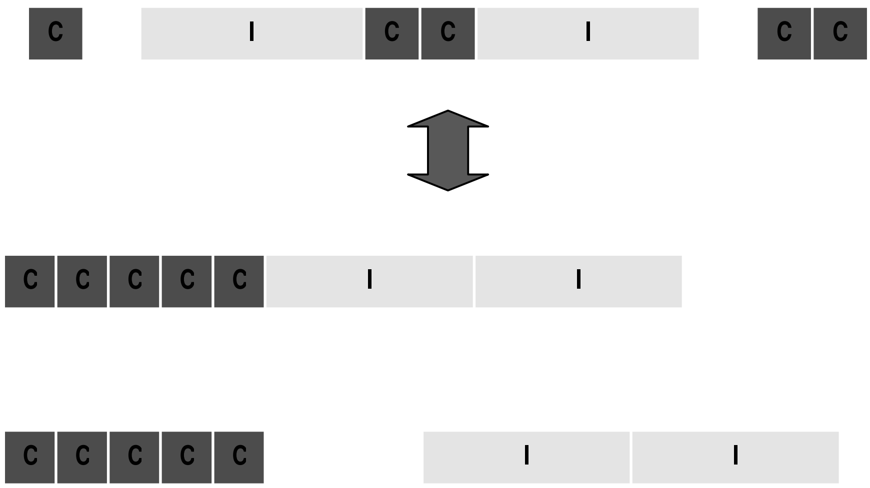

Figure 2. Compacting storage of re-ordered DDT.
Without padding, and with correct alignment.

4.1.3. FT-MPI DDT performance

Tests comparing the DDT code to MPICH (1.3.1) on a ninety three element DDT taken from a fluid dynamic code were performed between Sun SPARC Solaris and Red Hat (6.1) Linux machines as shown in table 1 below. The tests were on small and medium arrays of this data type. All the tests were performed using MPICH MPI_Send and MPI_Recv operations, so that the point to point communications speeds were not a factor, and only the handling of the data types was compared.

Type of operation (arch 2 arch) (method) (encoding)	11956 bytes B/W MB/Sec	% compared to MPICH	95648 bytes B/W MB/Sec	% compared to MPICH
Sparc 2 Sparc MPICH	5.49		5.47	
Sparc 2 Sparc DDT	6.54	+19 %	9.74	+78 %
Linux 2 Linux MPICH	7.11		8.79	
Linux 2 Linux DDT	7.87	+10 %	9.92	+81 %
Sparc 2 Linux MPICH	0.855		0.729	
Sparc 2 Linux DDT Byte Swap	5.87	+586 %	8.20	+1024 %
Sparc 2 Linux DDT XDR	5.31	+621 %	6.15	+ 743 %

Table 1. Performance of the FT-MPI DDT software compared to MPICH.

The tests show that the compacted data type handling gives from 10 to 19% improvement for small messages and 78 to 81% for larger arrays on same numeric representation machines. The benefits of buffer reuse and re-ordered data elements leads to considerable improvements on heterogeneous networks however. Noting that this test used MPICH to perform the point to point communication, and thus the overlapping of the data gather/scatter, encoding/decoding and non-blocking communication is not shown here, and is expected to yield even higher performance.

4.1.4. FT-MPI DDT additional benefits and future

The above tests were performed using the DDT software as a standalone library that can be used to improve any MPI implementation. This software is being made into a true MPI profiling library so that its use will be completely transparent. Two other efforts closely parallel this section of work on DDTs. PACX [19] from HLRS, RUS Stuttgart, requires the heterogeneous data conversion facilities and a project from NEC Europe [16] concentrates on efficient data type representation and transmission in homogeneous systems.

4.2. Collective Communications

The performance of the MPI's collective communications is critical to most MPI-based applications [6]. A general algorithm for a given collective communication operation may not give good performance on all systems due to the differences in architectures, network parameters and the storage capacity of the underlying MPI implementation [7]. In an attempt to improve over the usual collective library built on point to point communications design as in the logP model [9], we built a collective communications library that is tuned to its target architecture though the use a limited set of micro benchmarks. Once the static system is optimized we then tune the topology dynamically by re-orders the logical addresses to compensate for changing run time variations. Other projects that use a similar approach to optimizing include [12] and [13].

4.2.1. Collective communication algorithms and benchmarks

The micro-benchmarks are conducted for each of the different classes of MPI collective operations broadcast, gather, scatter, reduce etc individually. I.e. the algorithm that produces the best broadcast might not produce the best scatter even though they appear similar.

The algorithms tested are different variations of standard topologies and methods such as sequential, Rabenseifner [10], binary and binomial trees, using different combinations of blocking/non-blocking send and receives. Each test is varied over a number of processors, message sizes and segmentation sizes. The segmenting of messages was found to improve bi-section bandwidth obtained depending on the target network.

These tests produce an optimal topology and segment size for each MPI collective of interest. Tests against vendor MPI implementations have shown that our collective algorithms are comparable or even faster as shown in figure 3.

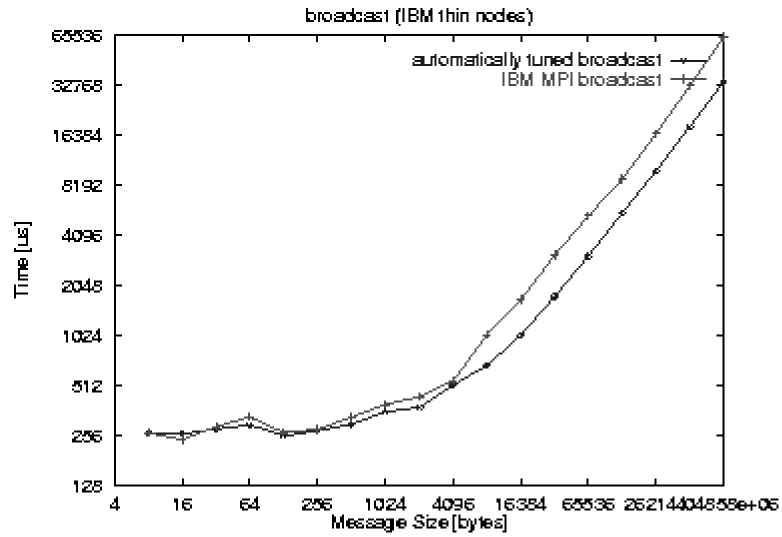

Figure 3. FT-MPI tuned collective broadcast verses IBM MPI broadcast on IBM SP2 thin node system.

4.2.2. Dynamic re-ordering of topologies

Most systems rely on all processes in a communicator or process group entering the collective communication call synchronously for good performance, i.e. all processes can start the operation without forcing others later in the topology to be delayed. There are some obvious cases where this is not the case:

(1) The application is executed upon heterogeneous computing platforms where the raw CPU power varies (or load balancing is not optimal).

(2) The computational cycle time of the application can be non-deterministic as is the case in many of the newer iterative solvers that may converge at different rates continuously.

Even when the application executes in a regular pattern, the physical network characteristics can cause problems with the simple logP model, such as when running between dispersed clusters. This problem becomes even more acute when the target systems latency is so low that any buffering, while waiting for slower nodes, drastically changes performance characteristics as is the case with BIP-MPI [14] and SCI MPI [8].

FT-MPI can be configured to use a reordering strategy that changes the non-root ordering of nodes in a tree depending on their availability at the beginning of the collective operation. Figure 4 shows the process by example of a binomial tree.

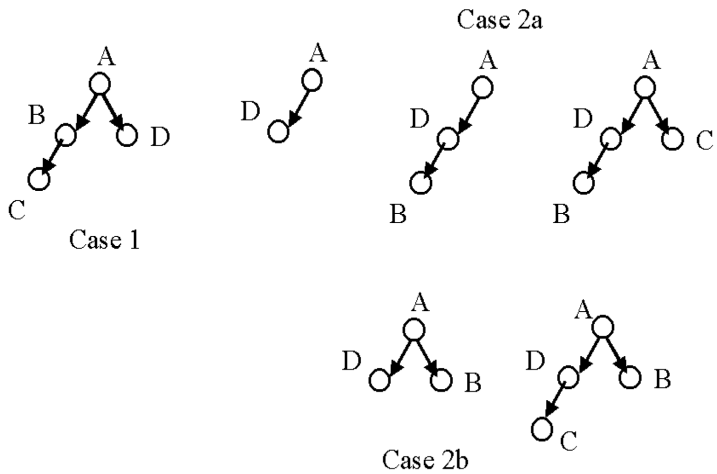

Figure 4. Re-ordering of a collective topology.

In Figure 4.1 Case 1 is where all processes within the tree are ready to run immediately and thus performance is optimal. In Case 2, both processes B and C are delayed and initially the root A can only send to D. As B and C become available, they are added to the topology. At this point we have to choose whether to add the nodes depth first as in Case 2a or breadth first as in Case 2b. Currently breadth first has given us the best results. Also note that in CASE 1, if process B is not ready to receive, it effects not only its own sub-tree, but depending on the message/segment size, it is possible that it would block any other messages that A might send, such as to Ds sub-tree etc. Faster network protocols might not implement non-blocking sends in a manner that could overcome this limitation without effecting the synchronous static optimal case, and thus blocking sends are often used instead.

4.3. Point to Point Multi-thread communications

FT-MPIs requirements for communications have forced us to use a multi-threaded communications library. The three most important criteria were:

- High performance networking is not effected by concurrent use of slower networking (Myrinet verses Ethernet)
- Non-blocking calls make progress outside of API calls
- Busy wait (CPU spinning) is avoided within the runtime library

To meet these requirements, in general communication requests are passed to a thread via a shared queue to be completed unless the calling thread can complete the operation immediately. Receives are placed into a pending queue by a separate thread. There is one sending and receiving thread per type of communication media. I.e. a thread for TCP communications, a thread for VIA and a thread for handling GM message events. The collective communications are built upon this point to point library.

4.4. Failure detection

It is important to note that the failure handler shown in figure 1, 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 this is usually due to direct communication with a failed party fails before the failed parties OS layer has notified other OS layers and their processes. 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.

5. OS support and the Harness G_HCORE

When FT-MPI was first designed the only Harness Kernel available was an experiment Java implementation from Emory University [5]. Tests were conducted to implement required services on this from C in the form of C-Java wrappers that made RMI calls. Although they worked, they were not very efficient and so FT-MPI was instead initially developed using the readily available PVM system.

As the project has progressed, the primary author developed the G_HCORE, a C based HARNESS core library that uses the same policies as the Java version. This core allows for services to be built that FT-MPI requires.

5.1. G_HCORE design and performance

The core is built as a daemon wrote in C code that 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) 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.

The Indirect invocation method allows a range of options such as:

- The H_GCOREs main thread calls the function directly
- The H_GCORE hands the function call over to a separate thread per invocation
- H_GCORE forks a new process to handle the request (once per invocation)
- H_GCORE forks a new handler that only handles that type of request (multi-invocation service)

Remote invocation services only provide very simple marsalling of argument lists. The simplest call format passes the socket of the request caller to the plug-in function which is then responsible for marshalling its own input and output much like skeleton functions under SUN RPC.

Currently the indirect remote invocation services are callable via both the UDP and TCP protocols. Table 2 contains performance details of the G_HCORE compared to the Java based Emory DVM system tested on a Linux cluster over 100Mbytes Second ethernet.

	Local (direct invocation)	Local (via TCP/Sockets to core)	Local (new thread)	Remote (RMI)	Remote (TCP)	Remote (UDP)
Emory Java DVM	-/-	10.4	0.172	1.406	8.6	-/-
G_HCORE	0.0021	0.58	0.189	-/-	1.17	0.32

Table 2. Performance of various invocation methods in milliseconds.

From Table 2 we can see that socket invocation under Java performs poorly, although RMI is comparable to C socket code for remote invocation. The fastest remote invocation method is via UDP on the G_HCORE at just over three hundred milliseconds per end to end invocation.

5.2. G_HCORE plug-in management

The plug-ins used by the G_HCORE can either be located locally via a mounted file system or downloaded via HTTP from a web repository. This loading scheme is very similar to that used by JAVA. The search actions are as follows:

- The local file system is checked first in a directory constructed from the plug-in name. I.e. Package FT_MPI, might have a component TCP_COMS. Thus the G_HCORE would first look in <HARNESS_ROOT>/lib/FT_MPI for a TCP_COMS shared object.
- If the plug-in was not in its correct location a search of the temporary cache directory would occur, i.e. <HARNESS_ROOT/cache/lib/FT_MPI. If the plug-in was found its time to live index would be checked to see if it was still current.
- If the plug-in was not local, then the internal system “get by HTTP” routine would be used. This currently functions with either a pure download, as in just a shared object stored in native format on a remote web server, or with a complex download that contains a PGP signed MIME encoded plug-in. This later method allows for signed plug-ins that are protected against external tampering once they are published.

The locations of the plug-ins available are stored within a distributed replicated database (DRD). This information is pushed to the database when plug-ins are ‘published’ at the individual web servers. The use of standard web servers for plug-in distribution was chosen to aid in individual site deployment of HARNESS, as existing servers can be used without modification.

5.3. G_HCORE services for FT-MPI

Current services required by FT-MPI break down into four categories:

- Spawn and Notify service. This plug-in 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.
- Naming services. These allocate unique identifiers in a distributed environment.
- Distributed Replicated Database (DRD). This service allows for system state and additional MetaData to be distributed, with replication specified at the record level. This plug-in has a secondary benefit as it can be used by the Emory DVMs PVM plug-in to implement the PVM 3.4 Mailbox features directly.

6. FT-MPI Tool support

Current MPI debuggers and visualization tools such as totalview, vampir, upshot etc do not have a concept of how to monitor MPI jobs that change their communicators on the fly, nor do they know how to monitor a virtual machine. To assist users in understanding these the author has implemented two monitor tools. HOSTINFO which displays the state of the Virtual Machine. COMINFO which displays processes and communicators in colour coded fashion so that users know the state of an applications processes and communicators. Both tools are currently built using the X11 libraries but will be rebuilt using the Java SWING system to aid portability. An example displays during a SHRINK communicator rebuild operation is shown in figures 5 to 7, where a process (rank 1) exits and the communicator is reduced in size and extent.

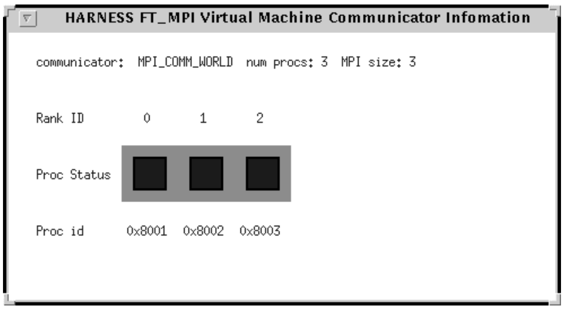

Figure 5. Cominfo display for a healthy three process MPI application. The colours of the inner boxes indicate the state of the processes and the outer box indicates the communicator state.

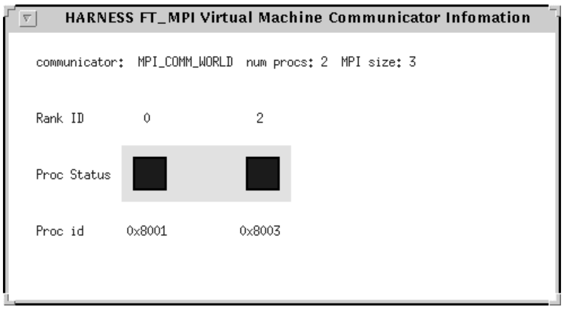

Figure 6. COMINFO display for an application with an exited process. Note that the number of nodes and size of communicator do not match.

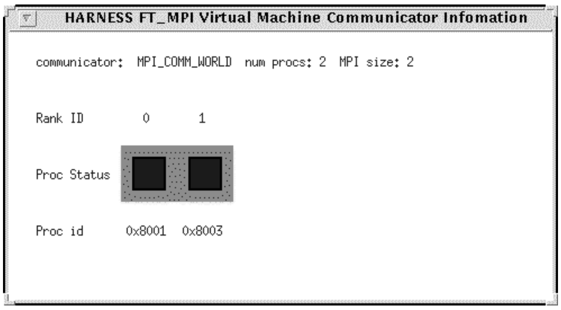

Figure 7. Cominfo display for the above application after a communicator rebuild using the SHRINK option. Note the communicator status box has changed back to a blue (dark) colour.

7. 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 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.

8. 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. Mauro Migliardi and Vaidy Sunderam, “PVM Emulation in the Harness MetaComputing System: A Plug-in Based Approach”, Lecture Notes in Computer Science (1697), pp 117-124, September 1999.
6. 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.
7. Thilo Kielmann, Henri E. Bal and Segei Gorlatch. Bandwidth-efficient Collective Communication for Clustered Wide Area Systems. *IPDPS 2000*, Cancun, Mexico. (May 1-5, 2000)
8. Lars Paul Huse, “Collective Communication on Dedicated Clusters of Workstations”, Proc of the 6th European PVM/MPI Users’ Group Meeting, Lecture Notes in Computer Science, Vol. 1697, Springer Verlag, pp. 469-476, Bareclona, September 1999.

9. David Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In Proc. Symposium on Principles and Practice of Parallel Programming (PpoPP), pages 1-12, San Diego, CA (May 1993).
10. R. Rabenseifner. A new optimized MPI reduce algorithm. http://www.hlrs.de/structure/support/parallel_computing/models/mpi/myreduce.html (1997).
11. Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker and Jack Dongarra. MPI- The Complete Reference. Volume 1, The MPI Core, second edition (1998).
12. M. Frigo. FFTW: An Adaptive Software Architecture for the FFT. Proceedings of the ICASSP Conference, page 1381, Vol. 3. (1998).
13. R. Clint Whaley and Jack Dongarra. Automatically Tuned Linear Algebra Software. SC98: High Performance Networking and Computing. http://www.cs.utk.edu/~rwhaley/ATL/INDEX.HTM. (1998)
14. L. Prylli and B. Tourancheau. “BIP: a new protocol designed for high performance networking on myrinet” In the PC-NOW workshop, IPPS/SPDP 1998, Orlando, USA, 1998.
15. 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.
16. Jesper Lasson Traff, Rolf Hempel, Hubert Ritzdort and Falk Zimmermann, “Flattening on the Fly: Efficient Handling of MPI Derived Datatypes”, Proc of the 6th European PVM/MPI Users’ Group Meeting, Lecture Notes in Computer Science, Vol. 1697, Springer Verlag, pp. 109-116, Bareclona, September 1999.
17. W.D. Gropp, E. Lusk and D. Swider, “Improving the performance of MPI derived datatypes”, In Third MPI Developer’s and User’s Conf (MPIDC’99), pp. 25-30, 1999.
18. Graham E Fagg, Kevin S. London and Jack J. Dongarra, "MPI_Connect, Managing Heterogeneous MPI Application Interoperation and Process Control", EuroPVM-MPI 98, Lecture Notes in Computer Science, Vol. 1497, pp.93-96, Springer Verlag, 1998.
19. Edgar Gabriel, Michael Resch, Thomas Beisel and Rainer Keller, “Distributed Computing in a Heterogeneous Computing Environment”, EuroPVM-MPI 98, Lecture Notes in Computer Science, Vol. 1497, pp.180-187, Springer Verlag, 1998.