FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world

Graham E. Fagg and Jack J. Dongarra

Department of Computer Science, University of Tennessee, 104 Ayres Hall, Knoxville, TN-37996-1301, 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 dynamic process operations would have effected their performance. As current HPC systems increase in size with higher potential levels of individual node failure, the need rises for new fault tolerant systems to be developed. Here we present a new implementation of MPI called FT-MPI[1] that allows the semantics and associated failure modes to be completely controlled by the application. Given is an overview of the FT-MPI semantics, design and some performance issues as well as the HARNESS g_hcore implementation it is built upon.

1. Introduction

Although MPI 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 simple 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 start 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 hense node failures.

[1] FT-MPI and HARNESS are supported in part by the US Department of Energy under contract DE-FG02-99ER25378.

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 checkpoint. FT-MPI is built on the HARNESS meta-computing system [1].

1.1 Check-point and roll back verse 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 Techischen Universitat Muchen 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 sycnronously, 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 their own complete version of MPI known as tuMPI as retro-fitting MPICH was considered too difficult.

Another systems 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 that 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 unpublished Implicit Fault Tolerance MPI project 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 a lost message in the event of a failure. This system appears only to support SPMD style computation and has a high overhead for every message.

2. 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. 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.

2.1.1 Failure modes

On detecting a failure within a communicator, that communicator is marked as having an (possible) 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.

2.1.2 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 shrank so that there are no holes in its data structures. The ranks of the processes are **changed**, forcing the application to recall MPI_COMM_RANK. A graphical example is given in section 6.

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 size of the communicator, not the number of valid processes within it.

REBUILD: Most complex in that it forces the creation of new processes to fill any gaps. The new processes can either be places in to the empty ranks, or the com-

municator can be shrank and the processes added 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 can not trap this, and only option is to change the communicator mode to one of the above modes.

Communications within the communicator are controlled by message mode for the communicator which is either:

NOP: No operations on error. I.e. no user level message operation 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 the error is available though the cached attributes interface in MPI.

2.1.3 Point to Point verse 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 only be the same as if there were no error, 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 the if the problematic nodes sends their data to the gatherer 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 [REF] 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 operation is in forcing an extra optimized barrier call within the collective call if required.

3. FT-MPI usage example

Typical usage of FT-MPI would be in the form of an error check and then some corrective action such as a communicator rebuild etc. A typical code fragment is shown below:

```
rc= MPI_Send (----, com);
If (rc==MPI_ERR_OTHER)
      MPI_Comm_dup (com, newcom);
      com = newcom;     /* continue.. */
```

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 for the PSTSWM 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, i.e. there is no C++ language support as of time of writing.

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 and collective communication is provided by the stand alone SNIPE_Lite communication library taken from the SNIPE project [4].

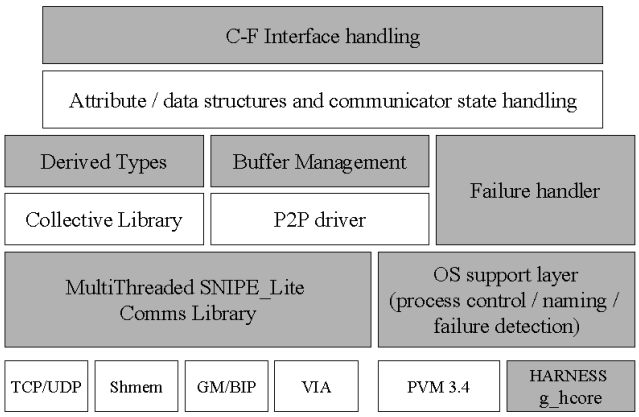

Fig. 1. Overall structure of the FT-MPI implementation.

A number of components have been extensively optimised, these include:

- Derived data types and message buffers. Particular attension has been paid in improving sparse data set and numeric representation handling.
- Collective communications. They have been tuned for both optimal topologies (ring verse binary vs binomial trees) as well as dynamic re-ordering of topologies.
- Point to point communication using a multithreaded SNIPE_Lite library that's allows separate threads to handle send and receives so that non-blocking communications still make progress while not within any MPI calls.

It is important to note that the failure handler gets notification of failures from both the 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 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.
The g_hcore library and daemon process (g_hcore_d) have good performance compared to the Java core especially in a LAN environment when using UDP, with remote function invocation times of 400uSeconds compared to several millisecond for Java RMI between remote JVMs.

Current services required by FT-MPI break down into three catagories:

1. Meta-Data storage. Provided by PVM in the form of message mboxes. Under the g_hcore as a multi-master master-slave replicated store.
2. Process control (spawn, kill). Provided using pvm_spawn and pvm_kill for PVM, and fork-exec and signal under the g_hcore_d.
3. Task exit notification. pvm_notify and pvm_probe under PVM, and via the spawn service under g_hcore catching sigchild and broken sockets.

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. Example

displays during a SHRINK communicator rebuild operation is shown in figures 2 to 4.

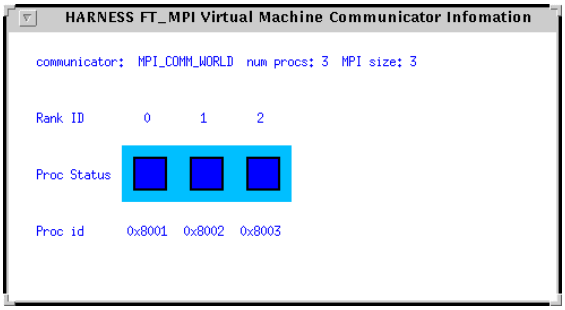

Fig. 2. Cominfo display for a healthy three process MPI application. The colours of the inner boxes indicate the state of the processes and the outer box indicates the communicator state.

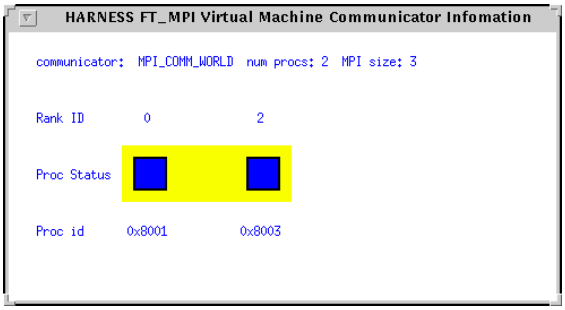

Fig. 3. Cominfo display for an application with an exited process. In this case the rank 1 process has exited. Note the communicator is maked as having an error and that the number of processes and size of the communicator are different.

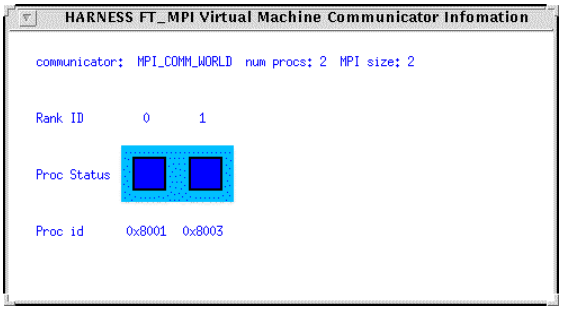

Fig. 4. Cominfo display for the above application after a communicator rebuild using the SHRINK option. Note the communicator status box has changed back to a blue (dark) colour.

7. Conclusions

FT-MPI is an attempt to provide application programmers with different methods of dealing with failure 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 for 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 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.