MPI/FT™: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing*

Rajanikanth Batchu[£§], Jothi P. Neelamegam[§], Zhenqian Cui[£],
Murali Beddhu[£], Anthony Skjellum[£§], Yoginder Dandass[§], Manoj Apte[§]

MPI Software Technology Inc[£]
101 S. Lafayette, Suite 33
Starkville, MS 39759 USA
{raj,murali,tony,zcui}@mpi-softtech.com

Mississippi State University[§]
Department of Computer Science
HPC Laboratory
{jothi,yogi,manoj}@cs.msstate.edu

Abstract

MPI has proven effective for parallel applications in situations with neither QoS nor fault handling. Emerging environments motivate fault-tolerant MPI middleware. Environments include space-based, wide-area/web/meta computing, and scalable clusters. MPI/FT, the system described here, trades off sufficient MPI fault coverage against acceptable parallel performance, based on mission requirements and constraints. MPI codes are evolved to use MPI/FT features. Non-portable code for event handlers and recovery management is isolated.

User-coordinated recovery, checkpointing, transparency and event handling, as well as evolvability of legacy MPI codes form key design criteria. Parallel self-checking threads address four levels of MPI implementation robustness, three of which are portable to any multi-threaded MPI. A taxonomy of application types provides six initial fault-relevant models; user-transparent parallel nMR computation is thereby considered. Key concepts from MPI/RT – real-time MPI – are also incorporated into MPI/FT, with further overt support for MPI/RT and MPI/FT in applications possible in future.

1. Introduction

The MPI standard has proven effective and sufficient for high-performance applications, in situations without either QoS or fault-handling requirements. COTS technology applied in relatively harsh as well as emerging environments motivates practical fault-tolerant MPI extensions, denoted MPI/FT here. These environments include deeply embedded, space-based, web/meta-computing, and production clustering. MPI/FT expands MPI in novel ways to include scope for fault/error detection and recovery. Process, node, and network failure are some of the faults that present themselves. Degree of user transparency, user-coordinated recovery, checkpointing services, and event handling, as well as evolvability of legacy MPI codes form key design criteria for MPI/FT. Parallel self-checking threads (SCTs) are introduced to address four levels of MPI implementation robustness, three of which are portable, that utilize the idea of "algorithmic-based fault tolerance" (ABFT) [1] as well as exploit voting on distributed, replicated state inside MPI itself. The MFT taxonomy of application-relevant models provides six specializations for parallel applications. MPI interoperation with nMR redundant computation in a user-transparent manner is considered. MPI/FT offers a practical architecture for trading off sufficient MPI fault coverage against acceptable parallel performance, based on actual mission and environmental requirements and constraints.

Briefly, here are areas where MPI applications can benefit from fault-handling capabilities:

- Space-based and similar constrained settings exist where single-event upsets (SEUs) cause transient failures.
- Node and network failures arising in long-running, scalable production systems.
- Computing on the Internet bears all the issues of intranet (or cluster) computing, plus the added transient issues of a complex, multi-owner, multi-site system.

The key demand on applications in such environments is that they run in a fail-through mode as opposed to fail-stop [2] mode that is typical in traditional environments. Here, fail-through mode is defined as the ability of an application to detect and recover from repeated failures, caused by extraneous events, until it completes successfully. In order to use legacy applications in such new environments and to facilitate the ease of

*Work performed in part with support from NASA under subcontract, 1219475, from the Jet Propulsion Laboratory, California Institute of Technology.

development of new applications one needs to develop additional support mechanisms/systems ("middleware") than what the traditional operating system can provide. Designing such a middleware suitable for scientific and other applications is the motivation of this paper.

Section 2 contains the background and related work. Section 3 presents the problem formulation, Section 4 introduces the self-checking MPI-layer and Section 5 introduces the various execution strategies. Parallel MPI is discussed in Section 6. Early experiments and preliminary results are presented in Section 7. Conclusions are presented in Section 8 with pointers for future work.

2. Background

The MPI [3] standards require that successful completion of an MPI application imply that all processes complete successfully, and that the default behavior in case of a process failure is the immediate termination of the application. MPI-1.2 allows users to attach an error-handling function to each communicator, which would be invoked in case of an abnormal return. However, performance constraints prevent MPI from detecting certain errors, and "catastrophic" errors may prevent MPI calls from returning to the caller, thereby preventing invocation of the user error-handler. MPI/Pro[4], MPICH [5], LAM [6] are some of the existing implementations of MPI, and none currently address fault issues.

The MPI/RT [7] standard is designed specifically to address issues related to Quality of Service (QoS), resource management and scheduling of communication tasks, which are not addressed in the other MPI's. Though the error handling capability of MPI/RT is much more sophisticated MPI-1.x and 2.x, it is still inadequate for fault-tolerance purposes. Notably useful to fault-tolerance is the Dynamic Process Management (DPM) capability of MPI-2 [8]. DPM is, however, insufficient to handle failures such as process crashes.

Cocheck [9] is among the earliest efforts towards incorporating a limited fault tolerance capability in MPI. It is a checkpointing library layered over MPI, and is an extension of the single process checkpointer of Condor [10] with a protocol for synchronous checkpointing. A single control process is used to send messages to all the processes to initiate checkpointing. This causes all the messages in transit to be flushed to arrive at a consistent global state after which the application is checkpointed. Cocheck uses a coarse-grain approach and is primarily developed and optimized for process migration. Limitations include scalability issues because of the control process and the high overheads associated with the flush protocol. Cocheck cannot consequently provide effective transient faults coverage.

The Starfish [11] environment for execution of static and dynamic MPI programs is based on the Ensemble [12] system. Starfish provides hooks to handle dynamic cluster changes and for checkpointing. It uses an event model where processes and components register to listen on events. This event bus provides messages reflecting changes in cluster configuration, and process failures.

FT-MPI [13] is a partial implementation of MPI-2 including DPM features, and uses a fault model that assumes fail-stop behavior of dying processes. FT-MPI concentrates on shrinking and growing communicators, based on the PVM model, which reflects terminations and creations of processes. Evripidou et al [14] use a similar approach where redundant processes are created a priori in order to replace processes that fail. This approach reduces complexity by having a fixed-size communicator.

3. Formulation

Here it is assumed that the application is well written and well tested so that abnormal behavior can be primarily attributed to an error/fault that doesn't originate in the application itself, (*e.g.*, a node failure). Such faults may manifest as application errors; for example, random bit flips causing an overflow of an integer. It is also assumed that the main memory and L2 cache are ECC protected and so are considered safe from random single bit errors.

The fail-through operation envisioned in this paper of such an MPI-application is shown in Figure 1. Typically, an MPI-application is started using the mpirun command. Once the mpirun starts the application, various scenarios are possible. In the event of a successful run the application would return success and then mpirun would return success. On the other hand, it is possible that the MPI-library might detect an error that would terminate the application, since all MPI errors are treated as fatal by default. Thus, the application would return unsuccessfully following which mpirun would return successfully. As a yet another possibility, one or more processes might crash which might lead to a hung application and thereby to a hung mpirun. A "hung" application is also possible even when all the processes are alive, in abnormal situations.

Since the user is mainly interested in the successful completion of the application, one needs to recover from various failures and continue operation until successful completion. When **mpirun** returns, one needs to determine whether the application was successful or not. If not, then application restart is needed, possibly from a checkpoint. When an application hangs, then one needs to detect this, to terminate the application, and then to restart in an appropriate, consistent manner.

In order to automate this process of guiding an application to successful completion, fault/error detection and recovery are needed at various stages. These stages are identified in Figures 1 and 2. MPI/FT is proposed here as the middleware/tool that incorporates the above

mentioned automation. To guard against data errors caused by random bit flips leading to scientific

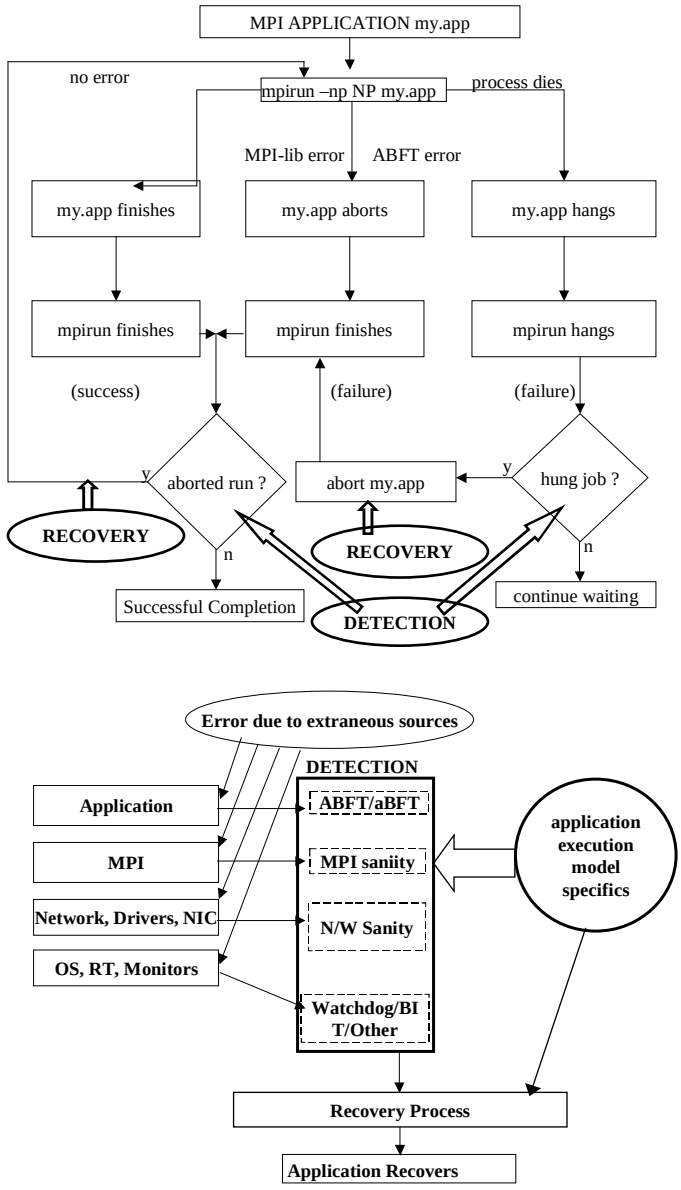

Figure 2. Detection and Recovery From Extraneously Induced Errors

inaccuracies of computed results, techniques such as ABFT have been used. Similarly, the concept of a self-checking MPI layer that seeks out inconsistencies in the internal state of MPI is introduced here.

Scientific programs mostly fall either under the category of a Master/Slave (M/S) model or under the category of an SPMD model. Thus, the current efforts of MPI/FT are focused in providing fault-tolerant coverage to these models. This approach leads to the identification of six different execution strategies as shown in Table 1. Central to all the six strategies is the idea of a Coordinator. For SPMD applications, the Coordinator is a separate entity that is transparent to the application. For M/S applications the root (master) node can itself be the Coordinator. The Coordinator will be run in an n-Modular Redundancy (nMR) mode whether the rest of the application itself is run in nMR mode or not. The disadvantage of the Coordinator is that it may not be scalable beyond an O(10) processors. The roles of the Coordinator are described in Section 5. All the models support user-assisted checkpointing.

4 Self-checking MPI layer

In addition to passing application-generated messages, the messaging layer can be used for many other purposes such as monitoring application progress, detecting inconsistencies in its internal state (for example, a corrupt communicator-table in one of the processes), propagating unrecoverable, local error information detected by various

Application Style	MPI Support	Model Name	nMR	Cp/Recov	
				App	Sys
SPMD	With MPI-1.2	MFT-I	No ranks nMR	Yes	Yes
		MFT-II	Several ranks nMR	Yes	Yes
	No MPI		No ranks nMR	Yes	
			Several ranks nMR	Yes	
Master/Slave	With MPI-1.2	MFT-IIIs	Rank 0 nMR	Yes	Yes
		MFT-IIIm	Several ranks nMR	Yes	Yes
	With MPI-2 DPM	MFT-IVs	Rank 0 nMR	Yes	Yes
		MFT-IVm	Several ranks nMR	Yes	Yes
	No MPI		No ranks nMR	Yes	
			Several ranks nMR	Yes	

Table 1. Overview of Models

layers globally, assisting in the recovery process, message-logging, responding to queries from other processes and so on. These additional tasks are collectively called self-checking tasks. Some of the self-checking tasks may be of a periodic nature in time and others may be in the nature of an interrupt service. The self-checking tasks need to be performed while in addition to transmitting application-generated messages. This requirement naturally leads to the idea of using additional threads, denoted the self-checking threads (SCTs), in order to perform the self-checking tasks.

Using SCTs the above mentioned tasks could be implemented so that SCTs would:

1) Vote on the global data structures periodically across processes.
2) Maintain multiple copies of local data and vote on them periodically.
3) Implement a non-blocking collective barrier with a timeout that would also be invoked periodically. The purpose of the non-blocking collective barrier is to detect failed processes.

4) Check queues for aging of messages.
5) Check for corrupt data structures, and performing other MPI consistency checks.
6) Monitor the health of internal dynamic memory allocation.
7) Watch the progress of the progress thread.
8) Watch the progress of messages between pairs of processes.

An SCT itself can be implemented in several ways: (i) a parallel user-level thread that allows any user program to access it, (ii) an internal thread visible to the MPI implementation only, (iii) both (i) and (ii). Design choices for an SCT vary from a trivially non-portable version to a non-trivially non-portable version with increasing functionalities and complexities as follows:

1) Trivially non-portable: Use existing internal data structures of a particular MPI implementation, and perform trivial/obvious checks on them. It is not visible to the user.
2) Trivially portable: (a) Use the PMPI profiling interface provided by the MPI standard to extend the previous approach across all MPI implementations. The complexity of operations that can be achieved is still trivial. It is visible to the user. (b) Adhere to the specifications of TotalView [15], and provide access to MPI internal structures across all MPI implementations. These structures include the send, and receive queue. It is visible to the user.
3) Non-trivially portable: This extends the functionality in (1), (2) and (3) by incorporating intelligence into the consistency checks. Can be visible to the user.
4) Non-trivially non-portable: This approach provides the most general functionality by defining new internal structures to aid consistency checks, etc. Such structures, and checks are specific to an MPI implementation.

As an example of adding "intelligence" to the error detection process, consider encoding the MPI header fields using Hamming codes. At a source this would prevent the message being sent to a wrong destination because of a transient error that corrupts the header while in the sending NIC. At a destination, this would prevent a potential non-delivery of messages to upper layers because of a mismatch of the source field between the incoming MPI header and the MPI call.

Since the data that passes through DMA engines is susceptible to corruption by extraneous sources, the data transfer at the MPI layer becomes unreliable even though the transfer at the data link level is reliable. To safeguard data transfer between MPI layers, either message-level CRC or time-based nMR could be used on MPI messages.

5. MFT Application Execution models

The various Middleware Fault Tolerance (MFT) Execution models are listed in Table 1. The roles of the Coordinator, introduced in Section 3, are as follows

1) Monitor the progress of the application.
2) Act as a virtual channel for messages. All messages from any rank to any other rank including itself will be routed transparently through the Coordinator which will maintain a log of the messages
3) Restart a failed process from a checkpoint and bring it to a consistent state with respect to the other ranks by replaying message logs to it and once it reaches a consistent state then allow the computations to continue in all ranks.
4) Send out periodic control messages to SCTs requesting information to be voted on.
5) Respond to information requests from SCTs.

MFT-I & II: SPMD WITH MPI

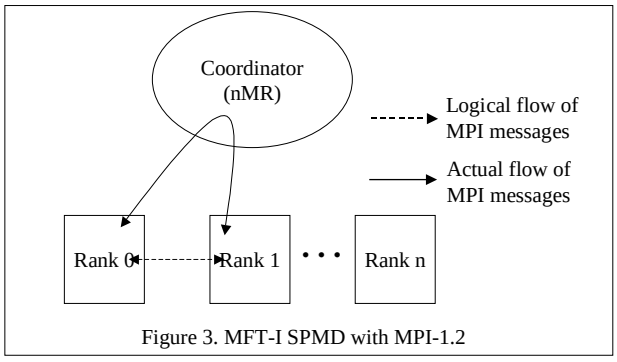

Figure 3. MFT-I SPMD with MPI-1.2

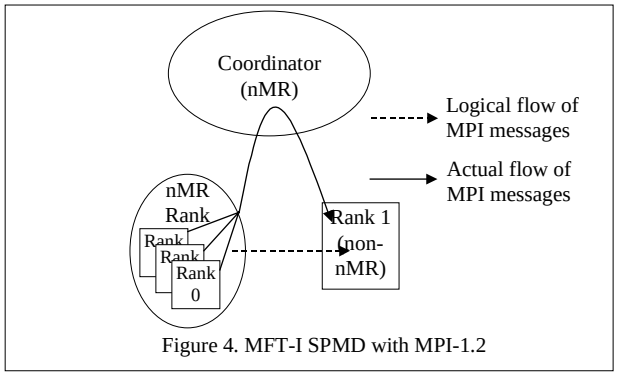

Figure 4. MFT-I SPMD with MPI-1.2

In MFT models I and II, shown in Figures 3 and 4, the Coordinator is transparent to the application. In model MFT-I, the application ranks are strictly in the simplex mode whereas in model MFT-II one or more of the application ranks would be run in the nMR mode. This feature would be useful if one or more ranks are more critical than others. These models work with MPI-1.2.

Note that as data transfer through the DMA engines could potentially be unreliable, one may choose to use MPI-

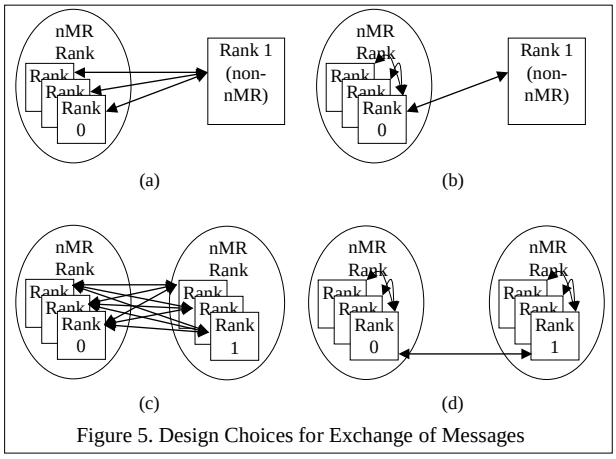

Figure 5. Design Choices for Exchange of Messages

level reliability measures as mentioned earlier even for the simplex mode. On the other hand, one may allow the data errors to pass through and be detected by the ABFT techniques. Similarly, for the nMR mode, where multiple copies of each message are generated by default, voting can take place either at the Coordinator or at the destination ranks depending upon the reliability of the MPI-level transfer. These choices are shown in Figure 5.

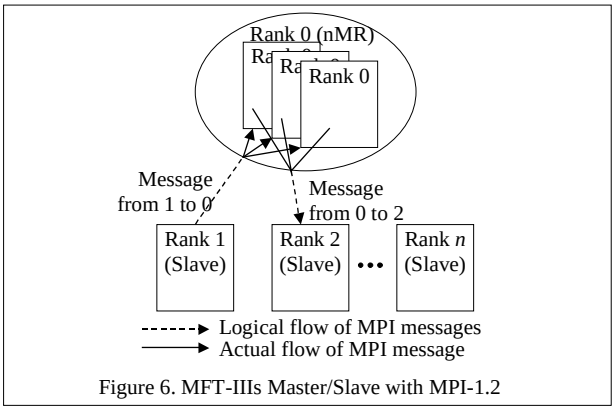

Figure 6. MFT-IIIs Master/Slave with MPI-1.2

MFT-IIIs & IIIm: MASTER/SLAVE WITH MPI-1.2

In these models, portrayed in Figure 6 and 7, the master plays the role of the Coordinator and hence is in nMR mode. The nMR nature of the master node would still be kept mostly transparent to the application. In Model MFT-IIIs, all the slaves are strictly in the simplex mode, whereas in Model MFT-IIIm, one or more slaves may be in nMR mode. Various choices for message exchanges exist as mentioned before and a trade-off study is underway. Since MPI-1.2 is used, the master node by itself cannot create a slave node in case a slave dies. This functionality, needed for efficient fail-through operations, must be provided by other middleware services. The recovery of only failed node(s) would be quicker compared to the case where one has to restart every node including the master each time a node fails.

MFT-IVs & IVm: MASTER/SLAVE WITH MPI-2

In Models MFT-IVs and MFT-IVm, the DPM capability of MPI-2 would be added to MPI-1.2 standard to permit the master node in nMR mode to dynamically re-spawn a

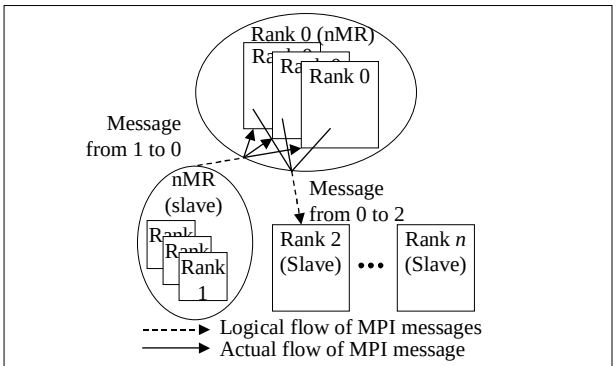

Figure 7. MFT-IIIs Master/Slave with MPI-1.2

slave node in case it dies during computation. In Model MFT-IVs, all the slave nodes will be run strictly in the simplex mode whereas in Model MFT-IVm, one or more slaves can be in the nMR mode. Sketches for Models MFT-IVs and MFT-IVm would be identical to Figures 6 and 7 and are not shown separately.

6. Parallel nMR

Consider a parallel execution in nMR mode using np processes. In this mode, one would create a set of n copies of an application with np number of ranks. Consider Figure 8, with n=3 and np=4 (chosen for the sake of illustration), where the three copies are named as A, B and C. For the sake of clarity only messages sent from node 0 to node 1 are shown. Moreover, this communication from node 0 to node 1 is shown from a sender's perspective using solid lines and from a receiver's perspective using dotted lines. The sender's perspective is shown only from node A0. Similar sender's perspectives from nodes B0 and C0 are not shown for clarity. Also elided are the receiver's perspectives for nodes A1 and B1. An MPI_Send call that used to send a message from rank 0 to rank 1 in the simplex mode would now send messages from each of the ranks A0, B0 and C0 to all of the ranks A1, B1 and C1 as shown in Figure 8. In other words, each of the nodes A1, B1 and C1 would receive messages from all of the nodes A0, B0 and C0. Nodes, A1, B1 and C1 would individually vote on the three messages they received and use the result of their vote on further computations. Thus, during normal computations, voting is needed only on the messages usually exchanged by a legacy application. This is in sharp contrast to the sequential-nMR mode where voting is needed on the entire computational state during each global iteration.

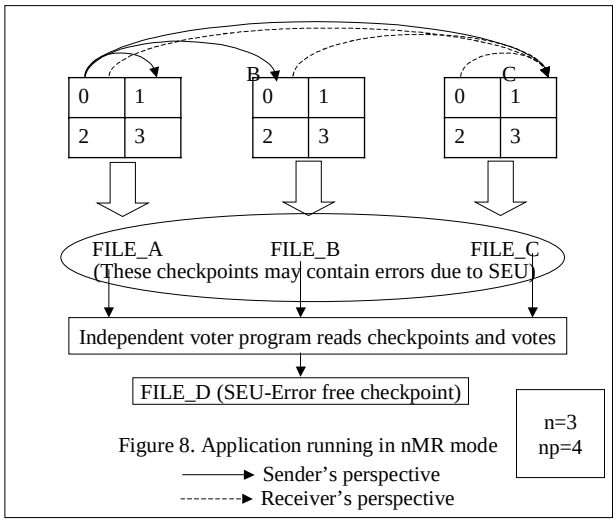

Figure 8. Application running in nMR mode

Another advantage of parallel-nMR mode is that local errors are contained within the nodes in which they occur. This is because voting would eliminate a single error (since n=3) at other nodes. As long as this single, local error remains non-fatal, the computation would continue. In the most general case, up to np local errors can exist simultaneously, subject to the condition that in every set of received messages only one message be erroneous. Thus, in Figure 8, one evident configuration of simultaneous faulty nodes that would not hinder the correct execution of the application is A0, A1, A2 and A3. An example of a possible configuration of faulty nodes that would lead to failure of the voting would be A0, B0, A2 and A3. Thus, parallel nMR can tolerate more local errors than the sequential nMR. However, every node failure increases the chances of the failure of the nMR mode. Thus, the allowable or tolerable number of node failures in the nMR mode before one falls back on a checkpoint depends upon a number of factors such as time between application checkpoints, rate of occurrence of the extraneous errors and ABFT error tolerance.

Since non-fatal local faults are confined to the local nodes in the parallel-TMR mode, voting on the entire computational state is unneeded until one wants to checkpoint the results, which is a big advantage over the sequential nMR method. In addition, if enough storage is available, one could store the results from all the nodes in which case voting on global data is never needed. Note that a restart from such a checkpoint would maintain the faulty nodes. Thus, whether one saves only one copy of the computation or n copies, it might be advantageous to do a global voting before checkpointing.

In the particular case of saving triplicate copies of the computation, one could do the global voting off-line while the application is running (*cf.,* Figure 8). Each of the three sets, A, B and C of the user program runs continuously and periodically checkpoints in triplicate to FILE_A, FILE_B and FILE_C respectively. A separate voter program then reads these files and votes on them and saves the output to FILE_D. Whenever there is a need to rollback to the previous checkpoint, all the three sets will read FILE_D. Alternately, in order to be strictly in the nMR mode, one could do this voting three times and create three copies, FILE_D1, FILE_D2 and FILE_D3 which will be read by, say, sets A, B and C respectively whenever there is need to rollback.

Finally, running an application in parallel-nMR mode may not significantly increase the computational time, assuming sufficient network resources. The overhead is in sending and receiving n-1 additional copies of each message and in voting on each set of n received messages. With high-speed networks such as the Myrinet, the cost of sending n-1 additional messages for each message is not expected to significantly affect the performance of applications over running them in simplex mode, especially for small values of n. The availability of enough computing nodes and potential power restrictions could possibly be the deciding factors to determine whether to run an application in nMR mode or not.

7. Early Experiments and Initial Results

There is considerable interest in using MPI/FT to provide fault-tolerant support to applications running in simplex mode in harsh environments replete with transient faults. In order to preserve the data integrity in messages, one could add a CRC to the entire message or use the time-based nMR technique. In order to compare the performance characteristics of the two approaches, the following timing study was conducted using two different MPI implementations, MPI/Pro (version 1.6.1-1tv) and MPICH (version 1.2.1).

For CRC messaging, the test constitutes the following operations: Calculate a 32 bit CRC and append it to the message and send the appended message to the receiver. After receiving the message at the receiver, compute the CRC again and compare with CRC included in the message. Then send an ACK to the original sender. The test is complete when the original sender receives the ACK. Sending and receiving was done using the blocking MPI_Send and MPI_Recv calls.

For time-based nMR messaging, the test constitutes the following operations: The sender sends the same message n times to the receiver. The receiver receives n copies of the message, buffers them and then votes on the n copies. After the voting is complete the receiver sends an ACK to the sender. The test is complete when the original sender receives the ACK. Sending and receiving was done using the blocking MPI_Send and MPI_Recv calls.

For all the results shown, the sizes of the message sent varied from 32 bytes to 512 kilobytes. For all the cases, the time reported is the total time it took for repeating the test 10,000 times.

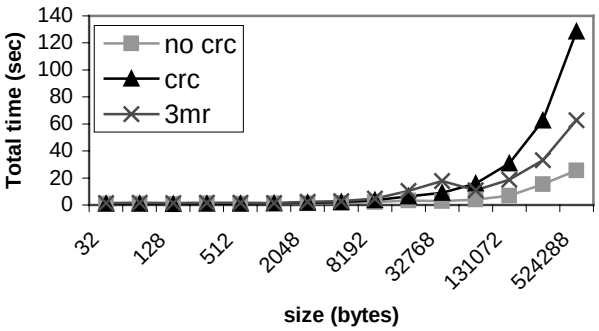

Figure 9. Comparison of nMR, CRC with No CRC using MPI/Pro

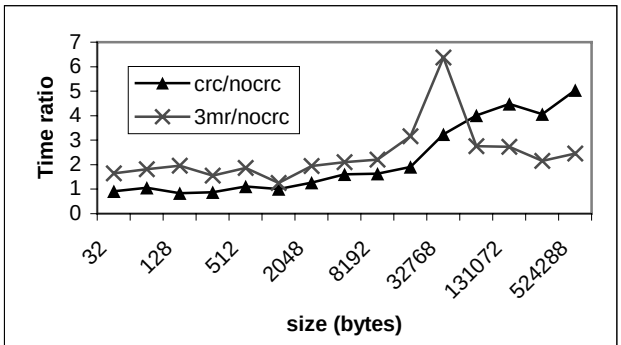

Figure 10. MPI/Pro Comparisons of Time Ratios

In Figure 9, total times for 10,000 runs using MPI/Pro are shown for the cases of nMR (with n=3), CRC and no-CRC. Note that for short messages, that is message sizes up to 32 KB, 3MR is more expensive that CRC. However, for long messages it takes more time to compute CRC than to send the message multiple numbers of times. Figure 10 magnifies the differences between CRC and time-based nMR messaging, shown in Figure 9, further. In this figure, the time ratio of the total time taken for CRC-messaging to the total time taken for no-CRC messaging is plotted, as is the ratio of the total time taken for 3MR messaging to the total time taken for no-CRC messaging. The spike at 32 KB in the 3MR/no-CRC ratios is an artifact of switching to long message protocol from short message protocol. Note that as seen in Figure 10, CRC-messaging takes virtually the same time as no-CRC messaging. However, as the message size increases, CRC takes more and more time compared to the no-CRC time. In fact, when the message size is 512 KB, CRC-messaging takes about 5 times the no-CRC time. On the other hand, nMR-messaging always takes more time than no-CRC messaging. However, even at message size 512 MB, it only takes about 2.5 times the no-CRC time, which is about two times faster than the CRC time.

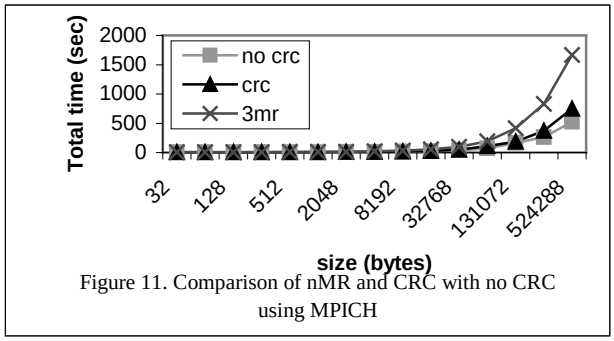

Figure 11. Comparison of nMR and CRC with no CRC using MPICH

Figure 11 shows a similar comparison as shown in Figure 9 with MPICH instead of MPI/Pro. Note that the ordinate in Figure 11 is an order of magnitude higher than in Figure 9. Also note that 3MR-messaging using MPICH is always more time consuming than CRC-messaging. In order to make the comparison between MPI/Pro and MPICH clearer, ratios of the time values shown in Figure 11 with the no-CRC time values shown in Figure 9 are obtained. Thus, the no-CRC time values of MPI/Pro are used as the basis. See Figure 12.

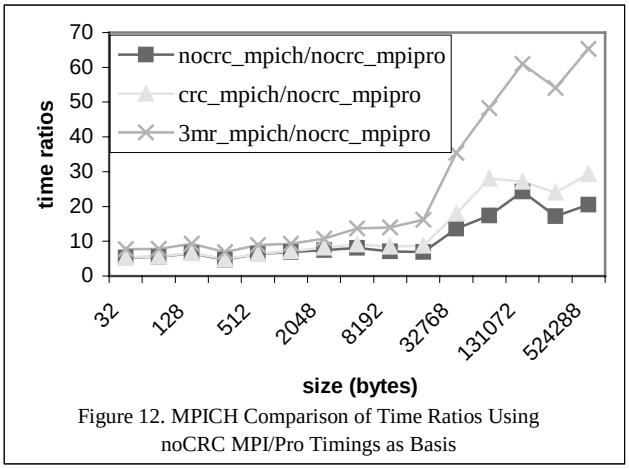

Figure 12. MPICH Comparison of Time Ratios Using noCRC MPI/Pro Timings as Basis

In Figure 13 and 14, MPI/Pro results are presented comparing measured timings for various nMR values as a function of message size. In Figure 13, actual time values are presented and in Figure 14 time ratios are presented using the MPI/Pro no-CRC result as the basis. As before the peaks occurring at the message size of 32 KB is an artifact of switching protocols from short message to long messages. Various interesting observations can be made from Figures 13 and 14. In Figure 13, note that as the value of n increases from 3 to 9, the time required to send and receive multiple copies also increase. It is interesting to see in Figure 14 that 9MR takes almost three times the time 3MR takes for short messages where as 9MR takes almost the same time as 3MR for long messages. This is because voting takes more time than network transit time for the network and CPU combinations considered. The fact that voting time dominates network transit time can

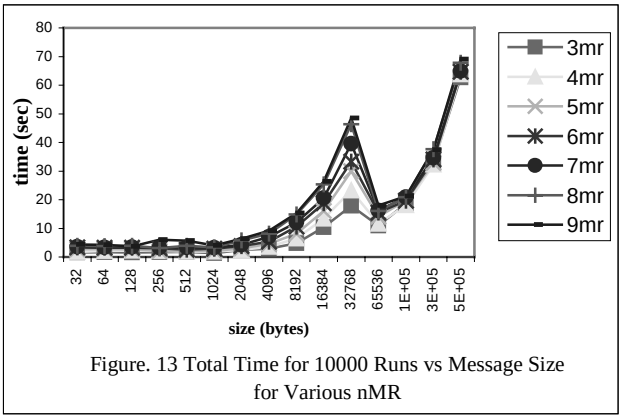

Figure. 13 Total Time for 10000 Runs vs Message Size for Various nMR

also be seen from observing in Figure 13 that timings for the pairs (4MR, 5MR), (6MR, 7MR) and (8MR, 9MR) are closer to each other than other values of nMR. This is so because in both 8MR and 9MR there are at least five message copies that must agree.

The results obtained for MPI/Pro indicate that both CRC-messaging and time-based nMR-messaging can play a role in different scenarios. For short message sizes CRC-messaging has a clear advantage. However, for long messages sizes, time-based nMR-messaging has the advantage over CRC-messaging. This advantage holds true even at 9MR levels, which is quite unexpected.

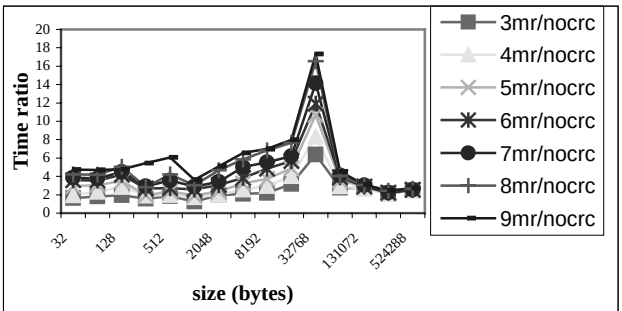

Figure 14. MPI/Pro Time Ratio Comparisons for various nMR

8. Conclusions

A fault-tolerant methodology leading to new MPI implementations is presented that provides support for successful completion of MPI applications in the presence of recurring, random, transient faults, induced extraneously. Depending upon the fault-tolerant coverage requirements, and other specifications and/or restrictions, the applications themselves can be run either in the simplex mode or in the parallel-nMR mode. Alternately, applications can be run in a mixed mode where critical nodes (as defined by the user) can be run in the nMR mode and non-critical nodes can be run in the simplex mode. The framework developed here supports message passing between all these possible configurations using a Coordinator which itself functions in the nMR mode. The present methodology introduces the idea of Self Checking Threads. The architecture presented is general and a restricted version of it that is suitable for applications running only in the simplex mode is currently being developed. A "shifted API" strategy for MPI is also being developed that would support additional fault recovery options for applications and would address evolvability. Future extensions include support for nMR mode. Practical use of MPI/FT is expected to be possible by end of 2001.

References

[1] S.J. Wang and N. K. Jha, "Algorithm-based fault tolerance for FFT networks," *In Proc. of Int'l Symp. on Circuits and Systems*, San Diego,CA, May 1992.

[2] R. D. Schlichting, and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems", *ACM Trans. on Computer Systems*, Vol. 1, No. 3, Aug. 1983, pp.222-238.

[3] http://www.mpi-forum.org/docs/mpi-11.ps .

[4] MPI Software Technology, Inc., MPI/Pro, 1999. http://mpi-softtech.com/.

[5] W. Gropp, E. Lusk, N. Doss, A. Skjellum, "MPICH: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", Parallel Computing, Vol 22, No. 6, Sep 1996, pp 789-828.

[6] G. Burns, R. Daoud, and J. Vaigl, " LAM: An open cluster environment for MPI," *In Proc. of Supercomp. Symp. 94*, Toronto, Canada.

[7] Real-Time message Passing Interface (MPI/RT) Forum. MPI/RT 1.0: Real-Time Message Passing Specification,1997. http://www.mpirt.org/drafts/mpirt-report-6mar00.ps

[8] Message Passing Interface Forum. MPI-2: Extensions to the Message Passing Interface, 1997. http://www.mpi-forum.org/docs/mpi-20.ps .

[9] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," *Proc. of the Int'l Par. Proc. Symp.*, IEEE Computer Soc. Press, Los Alamitos, C.A., 1996, pp. 526-531.

[10] T. Tannenbaum, and M. Litzkow, "Checkpointing and migration of Unix processes in the Condor distributed processing system," *Dr. Dobbs J.*, Feb. 1995, pp 40-48.

[11] A. Agbaria and R. Friedman, "Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations," *In Eighth IEEE Int. Symp. on High Perf. Dist. Computing*, 1999.

[12] M. Hayden, *The Ensemble System*. Doctoral dissertation, Cornell Univ., Dept. Computer Sciences, 1997.

[13] G.F. Fagg, and J.J. Dongarra, "FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World", *EuroPVM/MPI User's Group Meeting 2000*, Springer-Verilag, Berlin, Germany, 2000, pp. 346-353.

[14] P. Evripidou et al, " A Portable Fault Tolerant Scheme for MPI," *Proc. Int'l. Conf. on Par. and Dist. Proc. Techniques and Applications*, Las Vegas, N.V., 1998, pp 690-697.

[15] ETNUS Inc, TotalView User's Manual. Available from http://www.etnus.com/