Scalable Fault Tolerant MPI: Extending the recovery algorithm

Graham E. Fagg, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J. Dongarra

Dept. of Computer Science, 1122 Volunteer Blvd., Suite 413, The University of Tennessee, Knoxville, TN 37996-3450, USA

1 Abstract

Abstract. Fault Tolerant MPI (FT-MPI)[6] was designed as a solution to allow applications different methods to handle process failures beyond simple check-point restart schemes. The initial implementation of FT-MPI included a robust heavy weight system state recovery algorithm that was designed to manage the membership of MPI communicators during multiple failures. The algorithm and its implementation although robust, was very conservative and this effected its scalability on both very large clusters as well as on distributed systems. This paper details the FT-MPI recovery algorithm and our initial experiments with new recovery algorithms that are aimed at being both scalable and latency tolerant. Our conclusions shows that the use of both topology aware collective communication and distributed consensus algorithms together produce the best results.

2 Introduction

Application developers and end-users of high performance computing systems have today access to larger machines and more processors than ever before. Additionally, not only the individual machines are getting bigger, but with the recently increased network capacities, users have access to higher number of machines and computing resources. Concurrently using several computing resources, often referred to as Grid- or Metacomputing, further increases the number of processors used in each single job as well as the overall number of jobs, which a user can launch.

With increasing number of processors however, the probability, that an application is facing a node or link failure is also increasing. The current de-facto means of programming scientific applications for such large and distributed systems is via the message passing paradigm using an implementation of the Message Passing Interface (MPI) standard [10, 11]. Implementations of MPI such as FT-MPI [6] are designed to give users a choice on how to handle failures when they occur depending on the applications current state.

The internal algorithms used within FT-MPI during failure handling and recovery are also subject to the same scaling and performance issues that the rest

of the MPI library and applications face. Generally speaking, failure is assumed to be such a *rare* event that the performance of the recovery algorithm was considered secondary to its robustness. The system was designed originally for use on LAN based Clusters where some linear performance was acceptable at up to several hundred nodes, its scaling is however become an issue when FT-MPI is used on both larger MPP which are becoming more common and when running applications in a Meta/Grid environment.

This paper describes current work on FT-MPI to make its internal algorithms both scalable on single network (MPP) systems as well as more scalable when running applications across the wide area on potentially very high latency links.

This paper is ordered as follows: Section 3 detailed related work in fault tolerent MPIs, collective communication and distributed algorithms, section 4 details HARNESS/FT-MPIs architecture and the current recovery algorithm, section 5 the new methods togther with some initial experiment results (including transatlantic runs) and section 6 the conclusions and future work.

3 Related work

Work on making MPI implementations both fault tolerant and scalable can be split into the different categories based on the overall goals, either usually fault tolerance or scalablity [8] but rarely both. On the scalability front, related work includes both collective communication tuning and the development of distributed consensus algorithms though various schemes.

Most other fault tolerant MPI implementations support checkpoint and restart models, with or with various levels of message logging to improve performance. Coordinated checkpoint restart versions of MPI include: Co-Check MPI [12], LAM/MPI[14]. MPICH-V [4] uses a mix of uncoordinated check-pointing and distributed message logging. More relevant work includes: Starfish MPI [1] which uses low level atomic communications to maintain state and MPI/FT [2] which provides fault-tolerance by introducing a central co-ordinator and/or replicating MPI processes. Using these techniques, the library can detect erroneous messages by introducing a voting algorithm among the replicas and can survive process-failures. The drawback however is increased resource requirements and partially performance degradation. Finally, the Implicit Fault Tolerance MPI project MPI-FT [9] 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.

Starfish and MPI-FT are interesting in that they use classical distributed system solutions such as atomic communications and replication [17] to solve underlying state management problems.

4 Current Algorithm and Architecture of Harness and FT-MPI

FT-MPI was built from the ground up as an independent MPI implementation as part of the Department of Energy Heterogeneous Adaptable Reconfigurable Networked SyStems (HARNESS) project [3]. HARNESS provides a dynamic framework for adding new capabilities by using runtime plug-ins. FT-MPI is one such plug-in. A number of HARNESS services are required both during startup, failure-recovery and shutdown. This services are built in the form of plug-ins that can also be compiled as standalone daemons. The ones relevant to this work are:

- Spawn and Notify service. This service is provided by a plug-in which allows remote processes to be initiated and then monitored. The service notifies other interested processes when a failure or exit of the invoked process occurs. The notify message is either sent directly to all other MPI tasks directly or more commonly via the Notifier daemon which can provide additional diagnostic information if required.
- Naming services. These allocate unique identifiers in the distributed environment for tasks, daemons and services (which are uniquely addressable). The name service also provides temporary internal system (not application) state storage for use during MPI application startup and recovery, via a comprehensive record facility.

An important point to note is that the Spawn and Notify Service together with the Notifier daemons are responsible for delivering Failure/Death events. When the notifier daemon is used it forces an ordering on the delivery of the death event messages, but it does not impose a time bounding other than that provided by the underlying communication system SNIPE [7]. i.e. it is best effort, with multiple retries. Processes can be assumed to be dead when either their Spawn service detects their death (SIGCHLD etc), another MPI process cannot contact them or their Spawn service is unreachable.

It is useful to know what exactly the meaning of *state* is. The state in the context of FT-MPI is which MPI processes make up the MPI Communicator MPI_COMM_WORLD. In addition, the state also contains the process connection information, i.e. IP host addresses and port numbers etc. (FT-MPI allows processes the right to migrate during the recovery, thus the connection information can change and needs to be recollected). The contact information for all processes is stored in the Name Service, but during the recovery each process *can* receive a complete copy of all other processes contact information, reducing accesses to the Name Service at the cost of local storage within the MPI runtime library.

Current Recovery Algorithm The current recovery algorithm is a multistage algorithm that can be viewed as a type of conditional ALL2ALL communication based on who failed and who recovered. The aim of the algorithm is to build a

new consistent state of the system after a failure is detected. The algorithm itself must also be able to handle failures during recovery (i.e. recursive failures). The overall design is quite simple, first we detect who failed and who survived, then we recover processes (if needed), verify that the recovery proceeded correctly, build a new consistent state, disseminate this new state and check that the new state has been accepted by all processes. The following is a simple outline:

- State Discovery (initial)
- Recovery Phase
- State Discovery (verification if starting new processes or migration)
- New State Proposal
- New State Acceptance
- Continue State if accepted, or restart if not accepted

The current implementation contains the notion of two classes of participating processes within the recovery; *Leaders* and *Peons*. The leader tasks are responsible for synchronization, initiating the Recovery Phase, building and disseminating the new state atomically. The peons just follow orders from the leaders. In the event that a peon dies during the recovery, the leaders will restart the recovery algorithm. If the leader dies, the peons will enter an election controlled by the name service using an atomic test_and_set primitive. A new leader will be elected, and this will restart the recovery algorithm. This process will continue until either the algorithm succeeds or until everyone has died.

As mentioned previously the delivery of the death events is ordered but not time bounded. This is the reason why the *Initial* and *verification* State Discovery and New State Acceptance phases exist. The leader processes cannot rely on only the death events to know the current state of the system. In the case of bust failures, the death events may not all arrive at the same time. A consequence of this could be that the leader recovers only a single failed process and either completes the algorithm only to immediately have to restart it, or it discovers at the end of a recovery that the one of processes in the final state has also failed. The Acceptance phase prevents some processes from receiving the New State and continuing, while other processes receive a late death event and then restart the recovery process. This is essential as the recovery is *collective* and hence synchronizing across MPI_COMM_WORLD and must therefore be consistent.

Implementation of current algorithm Each of the phases in the recovery algorithm are implemented internally by the Leaders and Peons as a state machine. Both classes of processes migrate from one state to another by sending or receiving messages (i.e. a death event is receiving a message). The Leader processes store a copy of the state in the Name Service. This assists new processes in finding out what to do in the case that they were started after a particular state has already been initialized.

The State Discovery phase is implemented by the Leader telling all Peons that they need to send him an acknowledgment message (ACK). The Peons reply back to the Leader by sending a message which contains their current contact

information, thus combining two messages into one slightly larger message. The Leader then waits for the number of replies plus the number of failures (m) plus one (for themselves) to equal the number of total processes (n) in the original MPI_COMM_WORLD. As the recovery system does not have any explicit timeouts, it relies on the conservation of messages, i.e. no ACK or death event messages are lost.

The Recovery phase involves the leaders using their copy of the current state and then building a new version of MPI_COMM_WORLD. Depending on the FT-MPI communicator mode [6] used this could involve rearranging processes ranks or spawning new processes. The phase starts with the Leaders telling their existing Peons to WAIT via a short message broadcast. (This is an artifact left over from an earlier version that used polling of the Name Service). If new processes are started, they discover from the Name Service that they need to ACK the Leaders, without the Leaders needing to send the ACK requests to the new processes directly. After this the Leaders again perform a State Discovery to ensure that any of the new (and current) processes have not since failed. If no new processes are required, the Leaders build the new state and then move to the New State Proposal phase.

The New State Proposal phase is where the Leaders propose the new state to all other processes. During the Acceptance phase, all processes get the chance to reject the new state. This currently only occurs if they detect the death of a processes that is included in the new state otherwise they automatically accept it. The Leader(s) collect the votes and if ALL voted YES it sends out a final STATE OK message. Once this message has started to be transmitted, any further failures are IGNORED until a complete global restart of the algorithm by the Leader entering State Discovery phase again. This is implemented by associating each recovery with a unique value (*epoch*) assigned atomically by the Name Service. A Peon may detect the failure of a process, but will follow instructions from the Leaders, in this case STATE OK. The failure will however still be stored and not lost. This behavior prevents some processes from exiting the recovery while other processes continue to attempt to recover again.

Cost of current algorithm The current algorithm can be broken down into a number of linear and collective communication patterns. This then allows us to both model and then predict the performance of the algorithm for predetermined conditions such as a burst of m failures.

- Discovery phases can be viewed and as a small message broadcast (request ACK) followed by a larger message gather (receive ACK).
- Recovery phase is first a small message broadcast (WAIT) followed by a series of sequential message transfers between the Leader, Spawn & Notify service, Name Server etc to start any required new processes.
- New State Proposal phase is a broadcast of the complete state (larger message).
- New State Acceptance phase is a small message reduce (not a gather).
- OK State phase is a small message broadcast.

Assuming a time per operation of Top(n) where n is the participants (including the root), the current algorithm would take:

T$total$ = T$bcast_ack$(n-m) + T$gather_ack$(n-m) + T$bcast_wait$(n-m) + T$spawn$(m) + T$bcast_ack$(n) + T$gather_ack$(n) + T$bcast_state$(n) + T$reduce_accept$(n) + T$bcast_ok$(n)

As some message sizes are identical for a number of operations we can replace them with an operator based solely on their message size. Also if we assume that n¿m (which typically is true as we have single failures) we can simplify the cost to:

T$total$ = 4 T$bcast_small$(n) + 2 T$gather_large$(n) + T$spawn$(m) + T$bcast_large$(n) + T$reduce_small$(n)

The initial implementation for LAN based systems used simple linear fault tolerant algorithms, as any collective communication had to be fault tolerant itself. Take for example the initial ACK broadcast. The Leader only potentially knows who some of the failed tasks might be, but the broadcast cannot deadlock if unexpected tasks are missing.

Assuming the cost of spawning is constant, and that all operations can be performed using approximately n messages then the current algorithm could be further simplified if we consider the time for sending only small T$small$ or larger messages T$large$ to:

T$total$ = 5n T$small$ + 3n T$large$ + T$spawn$ or O(8n)+T$spawn$

5 New scalable methods and experimental results

The aim of any new algorithm and implementation is to reduce the total cost for the recovery operation on both large LAN based systems as well for Grid and Metacomputing environments where the FT-MPI application is split across multiple clusters (from racks to continents).

Fault Tolerant Tuned Collectives. The original algorithm is implemented by a number of broadcast, gather and reduce operations. The first obvious solution is to replace the linear operations by tuned collective operations using a variety of topologies etc. This we have done using both binary and binomial trees. This work was not trivial for either the LAN or wide area cases simply due to the fault tolerant requirements placed on the operations being able to handle recursive failures (i.e. nodes in any topology disappearing unexpectedly). This has been achieved by the development of self healing tree/mesh topologies. Under normal conditions they operate by passing message as expected in static topologies. In the case of a failures they use neighboring nodes to reroute the messages.

Figure 1 shows how the original algorithm performs when using various combinations of linear and tuned FT-collectives. As can be expected the tree based (Binary Tree (Tr) / Binomial Tree (Bm)) implementations out performed the Linear (Li) versions. The results also clearly show that the implementation is more sensitive to the performance of the broadcast operations (Bc) than the gather/reduce (Ga) operations. This is to be expected as the previous section

showed that algorithm contains five broadcasts verses three gather/reduce operations. The best performing implementation used a binary tree broadcast and a linear gather, although we know this not to be true for larger process counts [13].

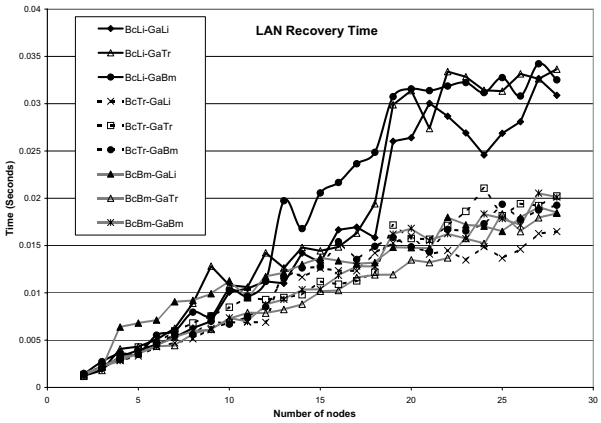

Fig. 1. Recovery time of the original algorithm when using various combinations of tuned collective operations

Multi-Zone Algorithms. For the wide area we have taken two approaches. The first is to arrange the process topologies so that they minimize the wide area communication, much the same as Magpie [16].

The second is to develop a multi-zone algorithm. This is where there is a leader process per zone. The lowest MPI ranked Leader becomes the master leader for the whole system. This leader synchronizes with the other zone leaders, who in turn execute the recovery algorithm across their zones. Unfortunately this algorithm does not benefit us much more than the latency sensitive topology algorithms due to the synchronizing nature of the New State Proposal and New State Acceptance phases, unless there are additional failures during the recovery.

Figure 2 shows how the original algorithm and the new multi-zone algorithm performs when executed between two remote clusters of various sizes. One cluster is located at the University of Tennessee USA, and the other is located at the University of Strasbourg France. A size of eight refers to four nodes at each site. The labels *SZ* indicate Single-Zone and *MZ* indicates Multi-Zone algorithms. Single-Zone *Linear* is the normal algorithm without modification, which performed badly as expected. Single-Zone *Tree1* is the single leader algorithm but using a binary tree where the layout of the process topology reducing the number of wide area hops. This algorithm performs well. Single-Zone *Tree2* is the single leader algorithm using a binary tree topology where no changes have been made to the layout to avoid wide area communication. Multi-Zone *Linear* is a multi

leader algorithm using linear collectives per zone, and Multi-Zone *OpColl* uses the best FT-tuned collective per zone. The two multi-zone algorithms perform best, although the node count is so low that it hides any advantage of the internal tree collectives within each individual zone. Overall the multi-zone tuned collective method perform the best as expected.

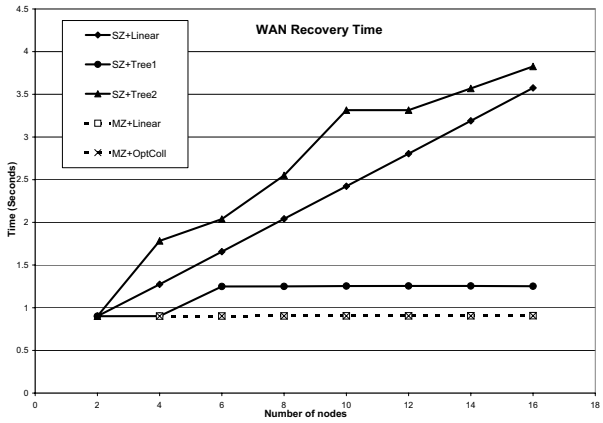

Fig. 2. Recovery time of various algorithms operating across a very high latency link

6 Conclusions and future work

The current FT-MPI recovery algorithm is robust but also heavyweight at approximately O(8n) messages. Although FT-MPI has successfully executed fault tolerant applications on medium sized IBM SP systems of up to six hundred processes its recovery algorithm is not scalable or latency tolerant. By using a combination of fault tolerant tree topology collective communications and a more distributed multi-coordinator (leader) based recovery algorithm, these scalability issues have been overcome.

Work is continuing on finding better distributed coordination algorithms and reducing the amount of state exchanged at the final stages of recovery. This is the first stage in moving FT-MPIs process fault tolerant model into the ultra scale arena. A latter stage will involve taking FT-MPIs recovery algorithm and adding it to the community Open MPI implementation [15].

Acknowledgments. This material is based upon work supported by the Department of Energy under Contract No. DE-FG02-02ER25536 and 8612-001-0449 through a subcontract from Rice University No. R7A827-792000. The NSF CISE Research Infrastructure program EIA-9972889 supported the infrastructure used in this work.

References

1. A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *In 8th IEEE International Symposium on High Performance Distributed Computing*, 1999.
2. R. Batchu, J. Neelamegam, Z. Cui, M. Beddhua, A. Skjellum, Y. Dandass, and M. Apte. Mpi/ft^{TM}: Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing. In *In Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid held in Melbourne, Australia.*, 2001.
3. Beck, Dongarra, Fagg, Geist, Gray, Kohl, Migliardi, K. Moore, T. Moore, Papadopoulous, Scott, and Sunderam. HARNESS:a next generation distributed virtual machine. *Future Generation Computer Systems*, 15, 1999.
4. G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov. MPICH-v: Toward a scalable fault tolerant MPI for volatile nodes. In *SuperComputing*, Baltimore USA, November 2002.
5. G. Burns and R. Daoud. Robust MPI message delivery through guaranteed resources. In *MPI Developers Conference*, June 1995.
6. G. E. Fagg, A. Bukovsky, and J. J. Dongarra. HARNESS and fault tolerant MPI. *Parallel Computing*, 27:1479–1496, 2001.
7. G. E. Fagg, K. Moore, and J. J. Dongarra. Scalable networked information processing environment (SNIPE). *Future Generation Computing Systems*, 15:571–582, 1999.
8. R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. Minnich, C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. In *ICS*. New York, USA, June. 22-26 2002.
9. S. Louca, N. Neophytou, A. Lachanas, and P. Evripidou. Mpi-ft: Portable fault tolerance scheme for MPI. In *Parallel Processing Letters, Vol. 10, No. 4, 371-382 , World Scientific Publishing Company.*, 2000.
10. Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, June 1995. http://www.mpi-forum.org/.
11. Message Passing Interface Forum. *MPI-2: Extensions to the Message Passing Interface*, July 1997. http://www.mpi-forum.org/.
12. G. Stellner. Cocheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, Honolulu, Hawaii, 1996.
13. S. S. Vadhiyar, G. E. Fagg, and J. J. Dongarra. Performance modeling for self-adapting collective communications for MPI. In *LACSI Symposium*. Springer, Eldorado Hotel, Santa Fe, NM, Oct. 15-18 2001.
14. Sriram Sankaran and Jeffrey M. Squyres and Brian Barrett and Andrew Lumsdaine and Jason Duell and Paul Hargrove and Eric Roman. The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In *LACSI Symposium*. Santa Fe, NM. October 2003.
15. E. Gabriel and G.E. Fagg and G. Bosilica and T. Angskun and J. J. Dongarra J.M. Squyres and V. Sahay and P. Kambadur and B. Barrett and A. Lumsdaine and R.H. Castain and D.J. Daniel and R.L. Graham and T.S. Woodall. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungry, 2004.
16. Thilo Kielmann and Rutger F.H. Hofman and Henri E. Bal and Aske Plaat and Raoul A. F. Bhoedjang. MagPIe: MPI's collective communication operations for clustered wide area systems. In ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), 34(8), pp131–140, May 1999.
17. Andrew S. Tanenbaum and Maarten van Steen. Distributed Systems: Principles and Paradigms, Prentice Hall, 2002.