Enhancing Fault Tolerance in MPI for Modern InfiniBand Clusters

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree

Master of Science in the Graduate School of The Ohio State University

By

Karthik Gopalakrishnan, B.E

Graduate Program in Computer Science and Engineering

The Ohio State University

2009

Thesis Committee:

Dr. D.K. Panda, Adviser

Dr. P. Sadayappan

ABSTRACT

There has been an unprecedented increase in the number of large scale computing clusters in the recent past. The advent of multi-core processors and high speed interconnects such as InniBand, which provide excellent performance at a reasonable cost, have contributed to this growth. However, the failure rate on these clusters has increased due to the increase in the number and scale of components. Thus, it has become vital for such systems to have fault tolerance capabilities.

Checkpoint / Restart and Job Migration are commonly used techniques for fault tolerance through failure recovery in computing clusters. Since MPI is the *de-facto* standard for parallel programming, it is an excellent candidate where these fault tolerance features can be implemented without exposing the complexity of the implementation to end user applications. Furthermore, failure detection and propogation of the fault information is an equally important topic of research in large peta-scale clusters.

In this thesis, we propose a design for a Checkpoint / Restart framework for MPI that facilitates the easy addition of new communication channels into the existing framework. Additionally, we also propose the use of the Fault Tolerant Backplane (FTB), provided by the Coordinated Infrastructure for Fault Tolerance Systems

(CIFTS), to implement Checkpoint / Restart and Job Migration mechanisms. We also design the InniBand and IPMI Fault Monitoring FTB components and demonstrate their use in detecting faults in the HPC system, and to trigger the Checkpoint / Restart or Job Migration mechanisms, as necessary. These designs are evaluated using a range of benchmarks and applications. The designs developed as a part of this thesis are available in MVAPICH2, a popular open source MPI implementation with almost a thousand active users.

Dedicated to my mother, Meera

ACKNOWLEDGMENTS

I would like to thank my adviser, Professor D. K. Panda for his guidance throughout the duration of my Masters. I also thank Professor P. Sadayappan for agreeing to serve on my Master's examination committee. He has always been a huge inspiration.

I would like to thank my colleagues, friends and family for their help, encouragement and support. I would also like to extend my gratitude to the staff in the CSE Department for making my stay at OSU a pleasant one.

I would like to specially thank Aditi and my mother for their immense motivation, without which, I would not have pursued my Masters.

VITA

2005	B.E., Electronics and Communication Engineering, Visvesvaraya Technological University, Belgaum, India.
2005-2007	Software Engineer, Hewlett-Packard India
2007-2009	Graduate Research Associate, The Ohio State University

PUBLICATIONS

Xiangyong Ouyang, Karthik Gopalakrishnan, Tejus Gangadharappa and Dhabaleswar K. Panda "Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture". *HiPC '09: Proceedings of the 2009 16th International Conference on High Performance Computing, December 2009*

Xiangyong Ouyang, Karthik Gopalakrishnan and Dhabaleswar K. Panda "Accelerating Checkpoint Operation by Node-Level Write Aggregation on Multicore Systems". *ICPP '09: Proceedings of the 2009 38th International Conference on Parallel Processing, September 2009*

Karthik Gopalakrishnan, Lei Chai, Wei Huang, Amith Mamidala and Dhabaleswar K. Panda "Efficient Checkpoint/Restart for Multi-Channel MPI over Multi-core Clusters". *OSU Technical Report. June 2009*

Gopal Santhanaraman, Pavan Balaji, Karthik Gopalakrishnan, Rajeev Thakur, William Gropp and Dhabaleswar K. Panda "Natively Supporting True One-sided Communication in MPI on Multi-core Systems with InfiniBand". *CCGrid '09: Proceedings of the 2009 9th International Symposium on Cluster Computing and the Grid, May 2009*

Matthew J. Koop, Wei Huang, Karthik Gopalakrishnan and Dhabaleswar K. Panda "Performance Analysis and Evaluation of PCIe 2.0 and Quad-Data Rate InfiniBand". *HOTI '08: Proceedings of the 2008 16th IEEE Symposium on High Performance Interconnects, August 2008*

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. D. K. Panda

TABLE OF CONTENTS

Page

LIST OF TABLES

Table **Page**

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Large scale compute clusters continue to grow to ever-increasing proportions. The Top500[15] list of high performance machines in the world indicates that the total core count of all the systems on the list as of June 2009 is almost ten times the core count as of June 2004. There has been an order of magnitude increase in the number of cores in the last four years. Clearly, this trend is expected to continue for many years to come.

The increase in the number of cores is mainly due to the advent of Multi-core processors. The performance gains that could be obtained in traditional single core processors by employing schemes such as frequency scaling and instruction pipelining was greatly diminished due to problems in power consumption, heat dissipation and fundamental limitations in exploiting Instruction Level Parallelism. Multi-core processors and the availability of commodity high speed interconnects such as InfiniBand[5] has resulted in the trend of using such large clusters.

Message Passing Interface (MPI) is the *de-facto* programming model used on such large scale clusters to write parallel applications. Most scientific applications that study Molecular Dynamics[3, 2], Finite Element Analysis[9], etc as well as Mathematical Libraries used in such applications[13, 14] are written in MPI.

The following section provides an overview of MPI and the features it provides.

1.1 Overview of Message Passing Interface

Message Passing Interface (MPI) is a language and machine independent communication paradigm that is predominantly used in parallel computing for inter-process communication. The MPI standard is defined by the MPI Forum[37]. MPICH2[10], MVAPICH2[11] and OpenMPI[29] are some of the popular MPI implementations.

The first version of the MPI standard (MPI-1), was defined in the early 90s. MPI-1 defined the basic point-to-point and collective communication interfaces. The second version of the MPI standard (MPI-2) was defined in 2003. MPI-2 introduced several extensions to the existing MPI-1 standard, such as the Dynamic Process Management interface, the Remote Memory Access interface and MPI File I/O interfaces.

In the following sections, we first provide a basic overview of the MPI communication model, Point-to-Point and Collective interfaces.

1.1.1 MPI Communicators

An MPI program is comprised of one or more processes, each of which can communicate with other processes. Every such MPI Process is identified by a [Rank, Process Group] tuple. The MPI Communicator encapsulates the process group information for the MPI process. All MPI operations are performed in the context of a communicator. MPI provides interfaces through which MPI processes can retrieve their ranks within a given communicator. All MPI communication primitives use the rank and the context information provided by a communicator to deliver messages

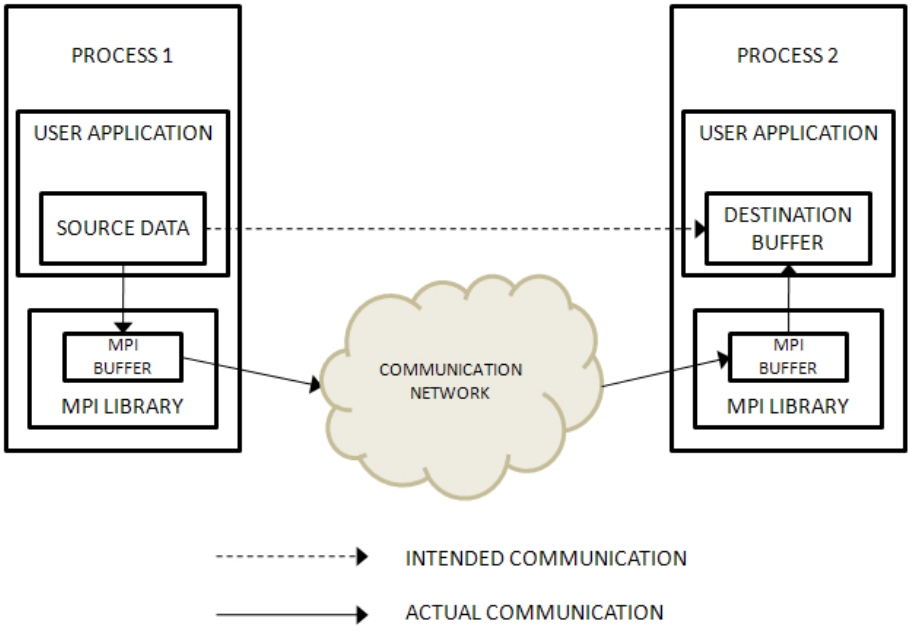


Figure 1.1: Overview of MPI

to the target. MPI_COMM_WORLD is a pre-defined communicator that facilitates communication among all processes within a MPI program.

1.1.2 Point-to-Point Communication

Two MPI processes can exchange messages using the Point-to-Point communication interface. In the Two-Sided Communication Model defined by MPI-1, the sender sends messages using a "Send" function. The "Send" function accepts the send buffer, the size and type of data being transmitted (CHAR, INT, FLOAT, etc), a tag for the data, and the receiver's rank and communicator. The receiver uses a "Receive" function to receive messages. The "Receive" function accepts the receive buffer, the size and type of data being received, the expected tag, and the sender's rank and communicator. The MPI_Send and MPI_Recv interfaces are used for blocking communication. These functions return only after the data transfer is completed. The MPI_Isend and MPI_Irecv interfaces are used for non-blocking communication. These functions initiate the data transfer and return immediately with a "Request Handle". This handle can be used to wait for the data transfer to complete using the MPI_Wait interface. In the One-Sided Communication Model defined by MPI-2, the source process can perform a "Put" or "Get" on the target process's "Window", without the involvement of the target. Every MPI process can create a window using the MPI_Win_create interface, which returns a "Window Pointer". The source can use this window pointer in the MPI_Put or MPI_Get interfaces, along with the target rank, buffer, size and type of data to transfer data to the target window.

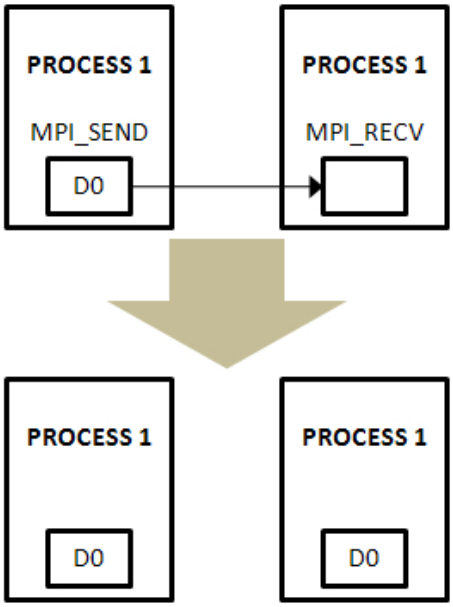


Figure 1.2: Point to Point Communication in MPI

1.1.3 Collective Communication

Collective operations are communication operations performed by a group of MPI processes that are part of a given communicator. The MPI standard defines several collective operations. MPI_Bcast is used to broadcast a certain message from one rank to all others. MPI_Alltoall is used by a set of processes to send data to all their peers and receive data from all the peers. MPI_Barrier, MPI_Reduce, MPI_Allreduce, MPI_Allgather are examples of other collectives operations.

1.2 Fault Tolerance in MPI

As computing clusters continue to increase in size, the Mean Time Between Failures (MTBF) for the cluster is diminishing rapidly. Studies[32] indicate that the MTBF for large scale clusters has reduced from days to a few hours. Many scientific MPI applications take anywhere from a few hours to a couple of days to complete their computation. Such applications whose average execution time exceeds the MTBF of the cluster can expect to see multiple failures during the lifetime of their execution. Thus, it becomes critical for such systems to be equipped with fault tolerance capability.

Although the MPI standard defines a rich set of primitives for data communication, the current MPI-2 specification does not provide any mechanisms for fault tolerance. As a result, most MPI implementations are designed without any support for fault tolerance. If the system experiences an error during the execution of a long running application, the application aborts and has to be executed from the

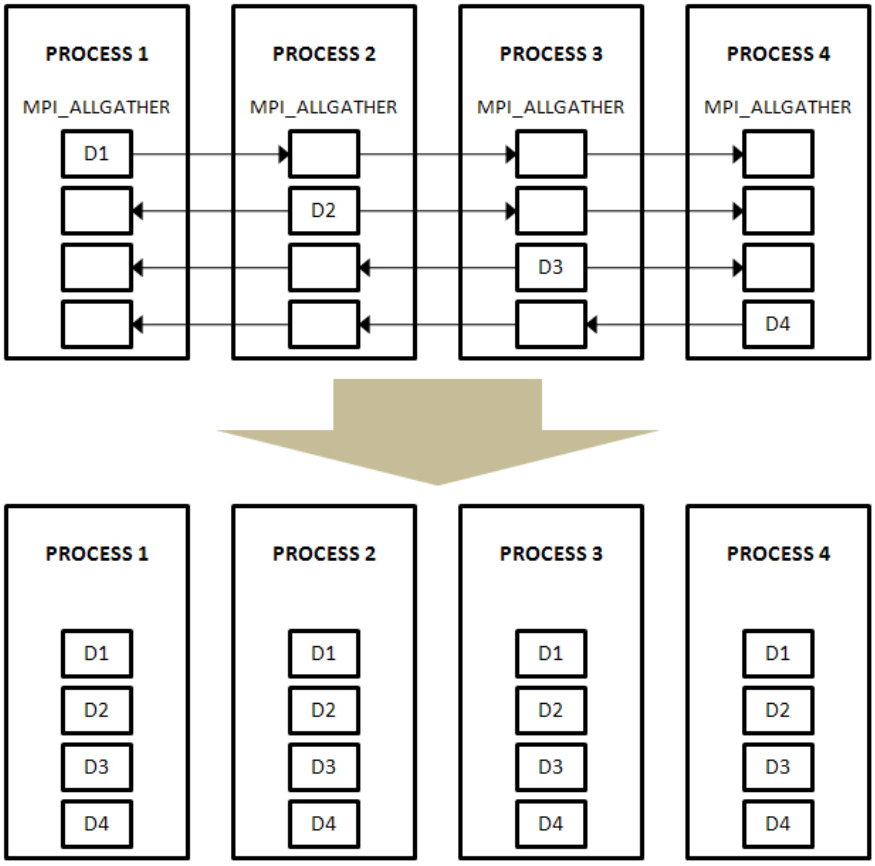


Figure 1.3: Collective Communication in MPI

beginning. A few MPI implementations [11, 29, 42] provide fault tolerance through rollback recovery using Checkpoint/Restart.

The following section provides a brief overview of Checkpoint/Restart.

1.2.1 Overview of Checkpoint/Restart

Checkpointing is the process of saving the state of a program at a given point of time during its execution, usually to stable storage, so that the program may be reconstructed at a later point in time. The process of reconstructing the program from a checkpoint is referred to as Restart.

Berkeley Lab's Checkpoint / Restart (BLCR) package[25] is a popular Checkpoint / Restart solution that is used by many MPI implementations. Checkpoint / Restart has many application in the context of High Performance Computing[24].

Multi-user Scheduling: HPC Clusters usually employ a job scheduler which enables multiple users to share the cluster's resources. Based on the scheduling policy used by the scheduler, there may be a necessity to preempt a long running job to run a shorter job that arrived much later. Checkpoint/Restart can be used to achieve this preemption without loss in computation time.

Application Migration: Well designed Checkpoint/Restart schemes allow processes to be checkpointed on one node and be restarted on another. This feature can be exploited to achieve process migration on computing clusters.

Application Backup: Checkpointing provides the backbone for fault tolerance through rollback recovery. An application maybe checkpointed periodically so that only the computation performed after the most recent checkpoint is lost in the event of

a failure. The rest of the discussion focuses on this application of Checkpoint/Restart.

1.2.2 Coordinated Infrastructure for Fault Tolerance Systems (CIFTS)

Considerable research has been conducted with respect to Fault Tolerance for system software, including the Message Passing Interface (MPI), Network Interconnects, File Systems, resource management infrastructure and applications. Most of the individual hardware and software components within the cluster implement mechanisms to provide some level of fault tolerance. However, these components work in isolation. This lack of system-wide fault tolerance features has emerged as one of the biggest problems on large HPC systems. To overcome this problem, a Coordinated Infrastructure for Fault Tolerance [1] has been designed to improve the level of fault-awareness within HPC systems.

The CIFTS Fault Tolerance Backplane[31] is an asynchronous messaging backplane that provides communication between the various system software components. The Fault Tolerance Backplane (FTB) provides a common infrastructure for the Operating System, Middleware, Libraries and Applications to exchange information related to hardware and software failures in real time. FTB can be used to tie various system components together as shown in Figure 1.4.

1.3 Problem Statement

Although some effort has been made towards providing fault tolerance to MPI based parallel programs using checkpointing and rollback recovery, almost all the work

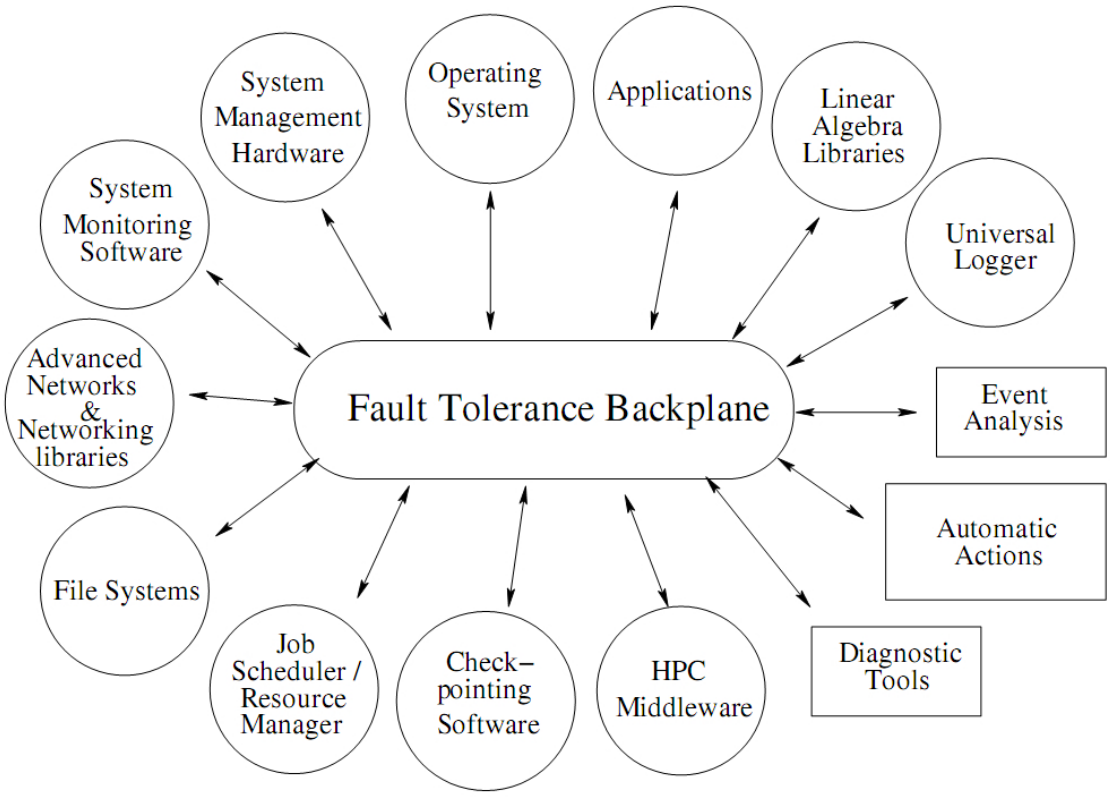


Figure 1.4: FTB Framework (Courtesy the CIFTS Team)

exclusively focus only on the network channel. The work in [40] focuses on providing Checkpoint/Restart capabilities for MPI programs communicating over TCP/IP. The work in [33] focuses on providing Checkpoint/Restart capabilities for MPI programs that use Myrinet. The work in [30] focuses on providing Checkpoint/Restart capabilities for MPI programs that communicate over InfiniBand. However, with the advent of multi-core processors, the performance of intra-node communication can be optimized in MPI implementations by using intra-chip and inter-chip operations. As a result, the solutions proposed in the works mentioned above is sub-optimal, since the intra-node shared memory channel cannot be used with Checkpoint/Restart.

Furthermore, as discussed in Section 1.2.2, most individual hardware and software components within the cluster work in isolation, without sharing information about the faults they encounter.

In this work, we try to address these shortcomings and specifically aim to answer the following questions:

1. How can we design a Checkpoint/Restart Framework for a Multi-channel MPI?

2. How can we design Checkpoint/Restart and Job Migration schemes for MPI using the CIFTS Framework?

3. How can we monitor the health of the InniBand network in the HPC Cluster, as well as the health of the HPC system as a whole?

4. How can we notify middle-ware about impending failures to take corrective action?

1.4 Organization of Thesis

The rest of this thesis is organized as follows. In Chapter 2, we present the design of a Checkpoint/Restart framework for MPI to checkpoint applications that use multiple communication channels. We also show that our design provides high performance as well as exibility with respect to process re-distribution after restart. In Chapter 3, we present the design for a Coordinated Fault Tolerance Framework for MPI using the Fault Tolerance Backplane. We design and evaluate schemes for Checkpoint/Restart and Job Migration using FTB. We also design the FTB-IB and FTB-IPMI Components that monitor the health of the InniBand Network and the HPC system as a whole. We present our conclusions and the direction of future work in Chapter 4.

CHAPTER 2

DESIGNING A MULTI-CHANNEL CHECKPOINT / RESTART FRAMEWORK FOR MPI

Ultra-scale computing clusters with high speed interconnects, such as InfiniBand, are being widely deployed for their excellent performance and cost effectiveness. However, the failure rate on these clusters increases along with increase in the number of augmented components. Thus, it becomes critical for such systems to be equipped with fault tolerance capability. The previous work[30] uses Checkpoint/Restart to save the state of the entire MPI job so that rollback recovery can be used to recover from a failure. However, this work only focuses on the network channel. MPI implementations optimize the performance of intra-node communication by using intra-chip and inter-chip operations. As a result, the solution proposed in the previous work is suboptimal for multi-core clusters since the intra-node shared memory channel cannot be used with Checkpoint/Restart.

In this chapter, we present the design for a framework to checkpoint multiple communication channels in MPI. The framework provides high performance as well as the flexibility to redistribute processes after restart. The design allows new communication channels to be easily incorporated into the framework.

The rest of the chapter is organized as follows: Section 2.1 provides the necessary background. Section 2.2 describes the framework. Section 2.3 discusses the design issues and challenges. Section 2.4 evaluates the performance of the design. Section 2.5 talks about the related work. Finally, we conclude and summarize the results in section 2.6.

2.1 Background

Checkpointing and rollback recovery is the most commonly used technique for system-level failure recovery in distributed systems. While various other techniques using application-level checkpointing[41, 20, 21] exist, system-level solutions are still popular since they are completely transparent to applications. A detailed comparison of various rollback recovery schemes can be found in [26]. Since MPI is the *de-facto* standard for parallel programming, it is the ideal place to integrate system-level failure recovery mechanisms and hide the complexity from end-user applications. Earlier studies have targeted various checkpointing and rollback recovery schemes including coordinated[30, 40] and uncoordinated checkpointing[17, 26]. In the context of modern clusters with high speed interconnects, coordinated checkpointing has its advantages. Uncoordinated checkpointing requires message logging, which adds considerable overhead when the amount of network traffic is significant. Furthermore, uncoordinated checkpointing is susceptible to the domino effect [39] where inter-process dependencies may result in all processes rolling back to the initial state.

In our previous work[30], we proposed a coordinated Checkpoint/Restart solution for MVAPICH2. Figure 2.1 shows the architecture that we proposed. It consists

of a global Checkpoint/Restart (CR) Manager, Local CR Managers, the InfiniBand Communication Channel Manager and the CR library.

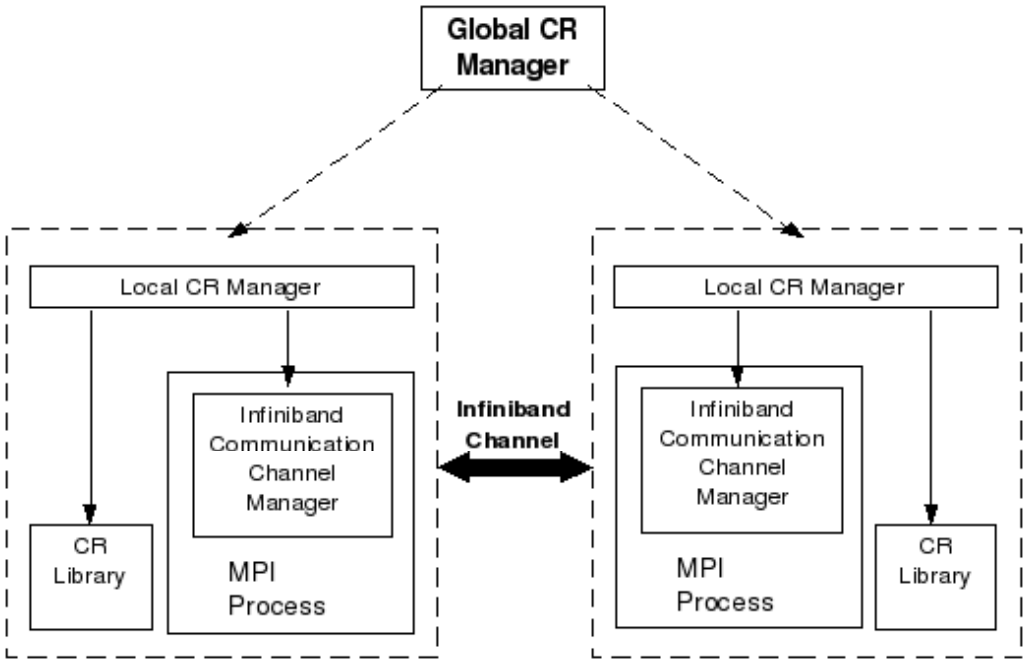


Figure 2.1: MVAPICH2 Checkpoint / Restart Architecture

When the user requests a checkpoint, a "checkpoint request" is generated by the global CR manager and is sent to the local CR managers which control the local MPI processes. The local CR managers inform the InfiniBand communication channel managers which is part of the MPI process. The InfiniBand communication channel manager then locks down the all the communication, including drain out and buffer all in-fly network messages and release the network resources. After that it gives a callback to the local CR manager to indicate that it is safe to take individual checkpoint. The local CR manager then invokes the CR library to checkpoint the

processes. Our implementation uses Berkley Lab's Checkpoint Restart package[28] as the the CR library. Once the process state has been locally checkpointed, the local CR manager informs the InfiniBand communication channel manager to reactivate the network communication. Similarly, to restart an MPI job, a "restart request" is sent from the global CR manager to all local CR managers. Upon receiving the request, the local CR managers launch the local MPI process through BLCR and reactivates the network by notifying the InfiniBand communication manager. As it can be seen, apart from saving local process state, a very important step of coordinated checkpoint is to handle the communication channels to make sure all individual checkpoints are taken at a consistent point.

However, this solution, like many other works on coordinated Checkpoint/Restart, selects a specific communication channel (InfiniBand in our, case) and forces all communication through this channel. This is not an efficient solution, since MPI implementations designed for multi-core clusters typically optimize communication primitives to take advantage of shared memory for intra-node communication, and a high speed interconnect for inter-node communication. The solution proposed in [30] would nullify this advantage. Efficient intra-node communication is not only important for point-to-point communication, but is also necessary to design highly efficient collective operations [36].

However, having multiple active communication channels during Checkpoint/Restart poses some interesting challenges which we describe in the following sections.

2.2 A Framework for Checkpointing Multi-Channel MPI

In this section we present a framework for checkpointing a multi-channel MPI library. Our approach is based on the coordinated checkpoint protocols. We first describe the design challenges, followed by the overall framework.

Although many open-source MPI library including [30, 40] provide Checkpoint / Restart support through coordinated protocols, few have addressed checkpointing MPI libraries which have multiple communication channels. To design an efficient checkpoint framework for multi-channel MPI, we need to achieve the following objectives:

- *Abstraction:* As we have described in Section 2.1, suspending and resuming communication is one of the key aspects for coordinated checkpointing. In the case of multiple communication channels, every communication channel needs to be suspended and resumed separately. As a result, not only the code becomes extremely complex due to handling different channels, it also becomes hard to extend the checkpoint functionality if new communication channels are designed. Our aim is to design a clear abstraction between the general suspend/resume functionality, which takes care of the coordination among peer processes, and the channel specific plug-ins, which provides the necessary functionality to checkpoint/restart individual communication channel.

- *Process re-distribution:* Checkpoint/Restart protocols achieve fault tolerance by checkpointing the computing processes and restarting them on new computing nodes if the old ones are malfunctioned. In this case, it is possible that the process topology will be changed after restart. For example, processes located

on the same nodes can be restarted on different ones. Processes communicate through shared memory, in this case, may have to use network after restart. Such process re-allocation poses additional challenges to checkpointing multi-channel MPI, whereas these design complexities may not be there if only one communication channel is used.

- *Collective Operations:* Collective operations optimized for multiple communication channel may also complicate the designs. For example, many collective operations can benefit from the more efficient intra-node shared memory communication[36] to minimize the inter-node traffic through network, achieving better performance. With these designs, however, collective operations are not simply built on top of point to point communication, but requires special coordination among peer processes. Thus, while many existing solutions can ignore additional complexities on collective operations, a multi-channel enabled design must carefully address such co-ordinations.

Our proposed framework is illustrated in Figure 2.2.

As it can be seen in the figure, we design a CR layer to perform all the generic checkpoint/restart functions. This CR layer will take care of the coordination with other parts of the system. For example, the CR layer is responsible to receive the checkpoint request from the job manager, notify each communication layer to suspend/resume the traffic, decide on a consistent time to issue local checkpoint operations using CR library (BLCR in our case). Many of these functions have been discussed in Section 2.1. Additionally, the CR layer will have to keep track of the process' location. For example, when two processes which were originally on the same node are restarted on different nodes, it should be able to detect this topology change

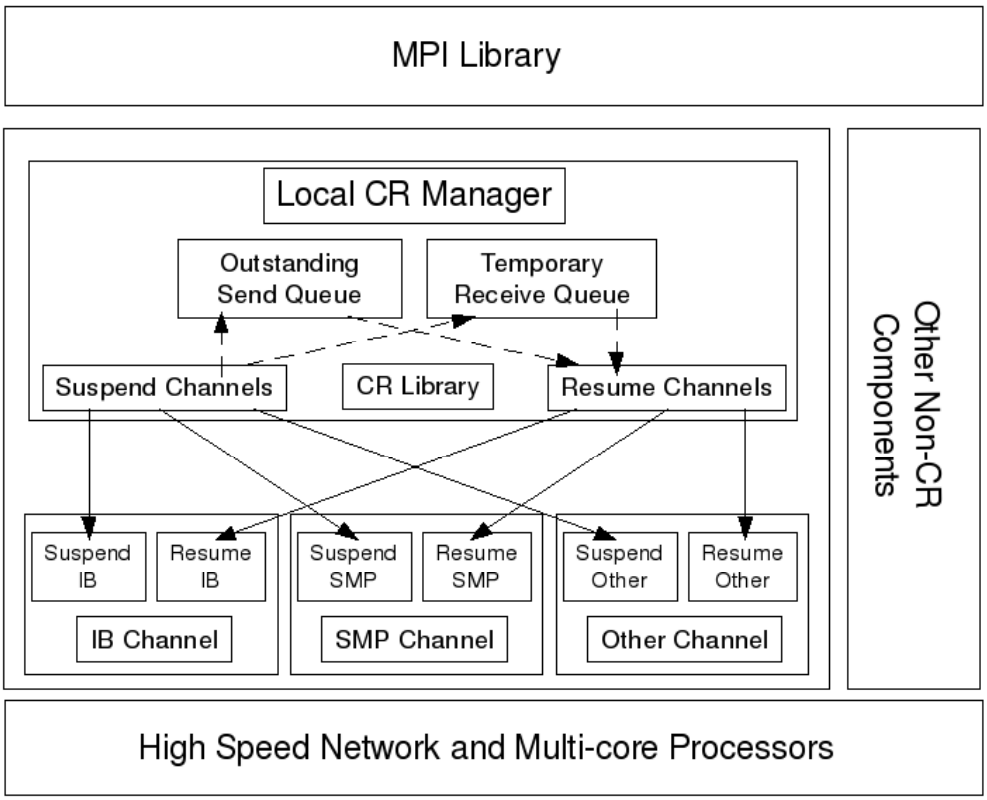


Figure 2.2: Checkpoint / Restart Framework for multi-channel MPI

and switch the communication channel accordingly from intra-node shared memory to network, and vice versa. To ensure in-order message delivery while switching the communication channels, the CR layer maintains two important queues, the outstanding send queue and the temporary receive queue. Once the checkpoint phase starts, all future send operations will be queued in the outstanding send queue. Meanwhile, all the channels drain in-flight messages and deliver them to the temporary receive queue. In this way, no message is left in any communication channel during the checkpoint. Hence, it is safe to switch communication channels after restart.

Every channel that needs to be suspended during the checkpoint phase will have to implement these functions and provide suspend/resume hooks to the CR layer through a standard interface. The CR layer will invoke these interfaces when necessary without having to worry about design details specific to the channel. In the next section, we will address some detailed design issues to suspend and resume communication traffic.

2.3 Detailed Design and Challenges

In this section, we will look at the detailed design of the checkpoint/restart framework for point-to-point and collective communication.

2.3.1 Point-to-point Communication

The new checkpoint/restart framework provides hooks that enables every communication channel to register callback functions with the framework. The callback functions of each of the communication channels are invoked by the framework during

the checkpointing process. The framework allows each channel to register two callback functions for the purpose of providing the ability to checkpoint point-to-point communication. The two functions, namely the "Suspend callback" function and the "Resume callback" function are discussed in greater detail in the following sections.

Suspend Callback function

The suspend callback function is invoked by the checkpoint/restart framework during checkpointing before the CR Library actually takes the checkpoint. It is the responsibility of this function to prepare the channel before a checkpoint. Since coordinated checkpointing is used, the suspend routine should ensure that no further sends are issued to this channel and that there are no messages in flight. This will ensure that all the processes are in a consistent state.

For the shared memory channel, the suspend callback function can be implemented to work in two phases.

Initial Synchronization Phase: In the initial synchronization phase, the callback function acquires a mutex that the main thread tries to acquire before every send operation. Once the callback function acquires the mutex, the main thread will not be able to issue any sends. Hence, the main thread is forced to wait on the mutex before proceeding with any communication.

Pre-checkpoint Coordination Phase: In the pre-checkpoint coordination phase, a "Suspend" control message is sent on the shared memory communication channel. Since the main thread can no longer perform sends and since the shared memory channel guarantees in order delivery of packets, reception of the Suspend message indicates that there are no pending messages on the communication channel. Hence, the channel can be marked as Suspended. Once all the processes have marked all

their channels as suspended, the shared memory region that was allocated for the send/receive buffers are released.

At this point, the CR library can save the state of the process to a file on the disk in the Local checkpointing Phase.

Resume Callback function

The resume callback function is invoked by the checkpoint/restart framework during checkpointing after the CR library has taken the checkpoint, as well as during restart after the CR library has restored the previously checkpointed process from disk.

For the shared memory channel, the resume callback function re-initializes the shared memory region and other necessary data structures that were destroyed by the suspend callback function. Once the data structures are restored, a "Resume" control message is then sent on the Shared Memory Channels to mark them as Active. The callback function then releases the mutex that the suspend callback function had acquired. Once the mutex is released, the main thread that was waiting on it now acquires the mutex.

At this point, the state of the system is identical to what it was before the Suspend callback was invoked. Hence, communication on all the channels proceed normally.

2.3.2 Challenges While Checkpointing Collective Operations

In the previous sections, we have seen how the checkpoint/restart framework handles point-to-point communication. The same designs are not sufficient to guarantee correctness of collective communication for the following reasons:

- Collective operations are invoked by multiple processes. Hence, group synchronization methods are required for coordination.

- Process skews can easily result in deadlocks as the checkpoint request may arrive at different phases across multiple processes making locking/unlocking complex to handle.

The rest of the section deals with the solution to these issues.

Synchronization and Consistency Mechanisms: It is imperative to guarantee consistency across all the processes taking part in the collective operation. This may not always be straightforward for collectives. For example, a typical shared memory collective operation consists of the following three phases:

- Intra-node communication via shared memory

- Inter-node communication via network point-to-point channels, and

- Intra-node communication for the final part of the algorithm

The point-to-point scheme described in the previous section is sufficient to ensure consistency of the processes when a checkpoint request arrives during the inter-node communication phase of all the processes. However, a different protocol is necessary when the request arrives during the intra-node communication phase of one of the processes. This is described in the following section.

Avoiding Deadlocks: Unlike the point-to-point scheme, where the Checkpoint / Restart framework just invokes the callbacks registered by each channel, the collectives explicitly have to notify the framework when it is all right to proceed with the checkpointing, after the processes have been synchronized. Due to this two way

communication between the collectives and the framework, there are possibilities of deadlock. Hence, special care has to be taken while designing the locking mechanism to avoid such scenarios.

The following section discusses the framework for checkpointing collective operations and the specifics of the implementation for the shared memory collectives.

2.3.3 Checkpointing Collective Operations

Figure 2.3 shows the overall operation of the collectives checkpointing. Due to the nature of the collective operations, the semantics of the callback functions necessary to implement checkpoint/restart are significantly different from those required for point-to-point communication. To be able to checkpoint collective operations, two callback functions, namely the "Request to Checkpoint callback" and the "Checkpoint Complete callback" are introduced. These functions are discussed in detail in the following sections.

Request to Checkpoint Callback function

The Request to Checkpoint callback function (RTC) is invoked by the local CR manager when a checkpoint is requested to be taken. This call notifies the collectives about the checkpoint request and returns immediately. Once the call returns, the local CR manager waits for a notification from the collectives indicating that they are ready to be checkpointed.

As discussed in the previous section, a different synchronization protocol is needed during the intra-node communication phase. We use the following protocol in our implementation.

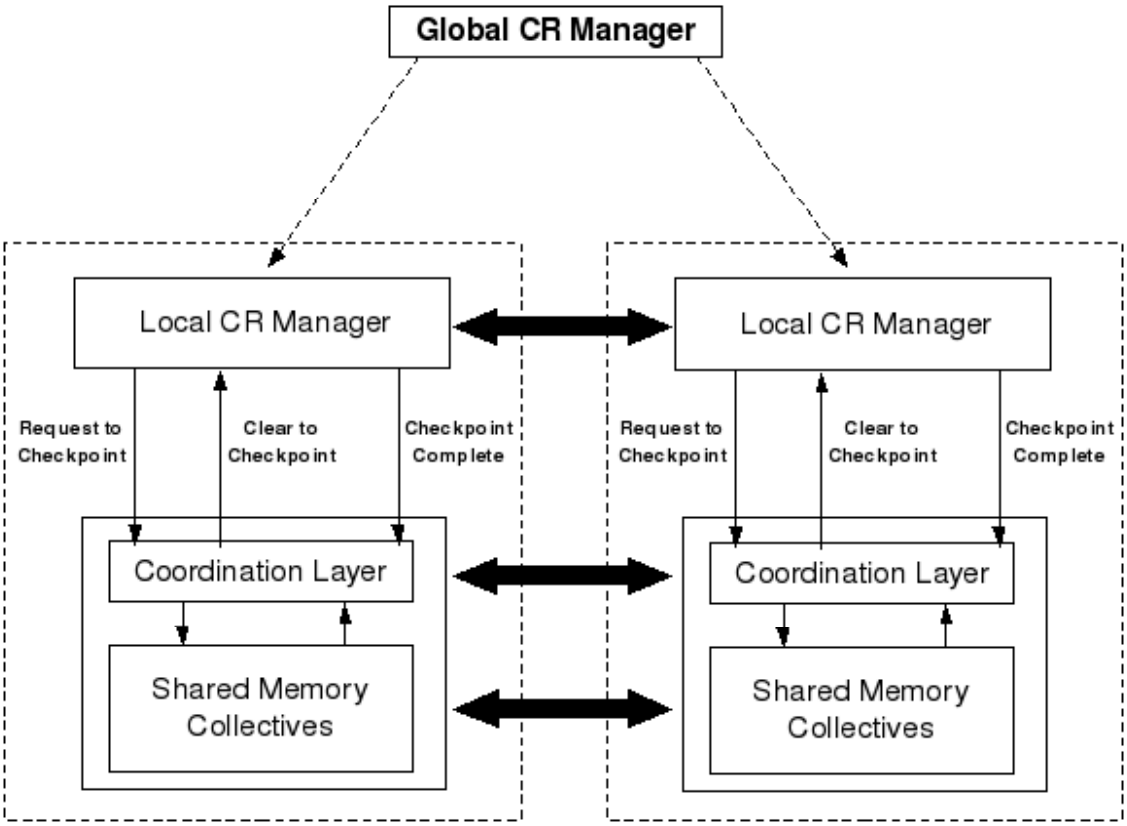


Figure 2.3: Checkpointing protocol for Shared Memory Collectives

The processes involved in the intra-node operation stop their communication and atomically increment a special field in the shared memory region. Each process continues to examine this field after incrementing it. Once the count becomes equal to the number of processes on local node, it indicates that all processes have stopped collective communication. At this point, all the local processes are in a consistent state. A leader is now chosen to copy the contents of the shared memory region to a local buffer and tear down the collectives' shared memory region. Once this is done, all the processes on the node call CTC and wait for checkpoint completion.

The notification is achieved thorough the "Clear to Checkpoint" function (CTC) whose pointer is passed down to the collectives during the RTC Call. The collective calls CTC when it is ready to be checkpointed. The checkpoint/restart framework proceeds with the checkpointing after receiving the CTC.

The checkpoint/restart framework now invokes the CR library to take a checkpoint.

Checkpoint Complete Callback function

Once the checkpoint has been taken, or when the processes have been restarted from a previously taken checkpoint, the framework invokes the Checkpoint Complete callback function (CC) to indicate that the checkpoint/restart operation has completed.

When CC is invoked, processes in a collective operation that were waiting for the checkpoint to complete are activated. The leader process initializes the collectives' shared memory region and restores the data structures that it had saved in its local buffer. Once the leader completes creating the shared memory region, all processes return from the callback.

At this point, the state of the system is identical to what it was before RTC was invoked. Hence, communication on all the channels proceed normally.

2.4 Performance Evaluation

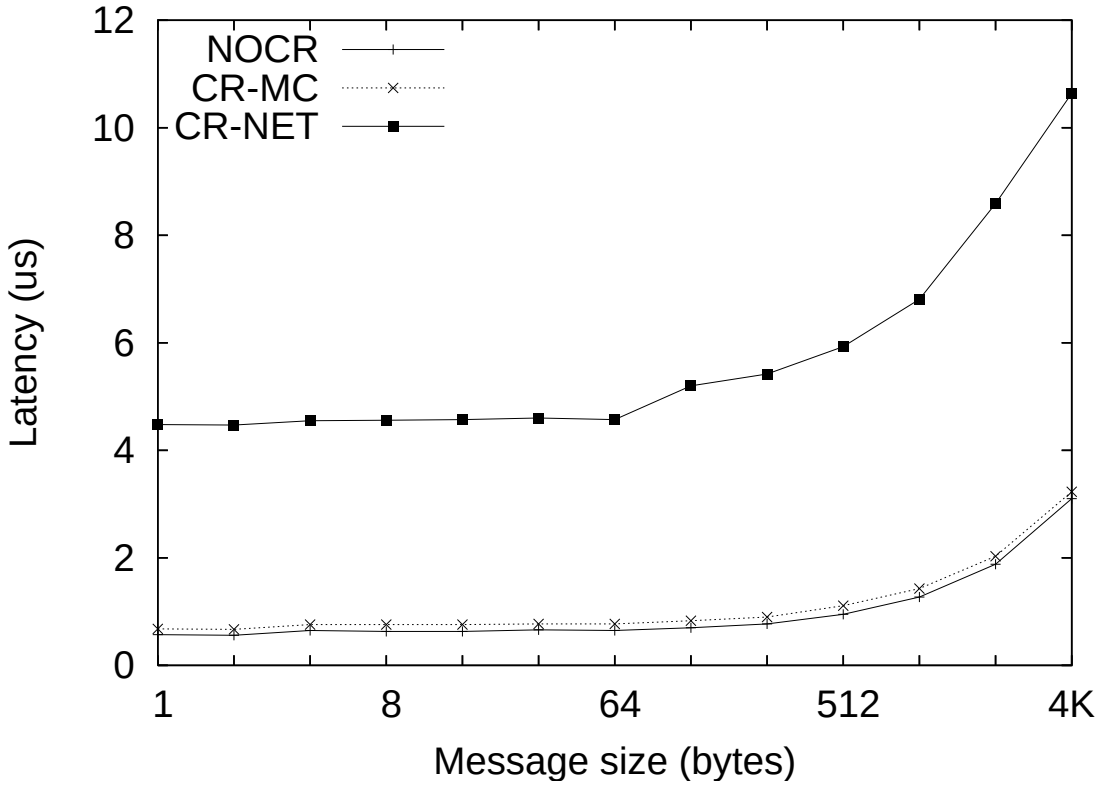


Figure 2.4: Performance of MPI Latency

In this section, we evaluate and analyze the performance of the proposed design using point-to-point, collective, and application level benchmarks. In Sections 2.4.1, 2.4.2, and 2.4.3, the experiments were conducted without taking the checkpoints to

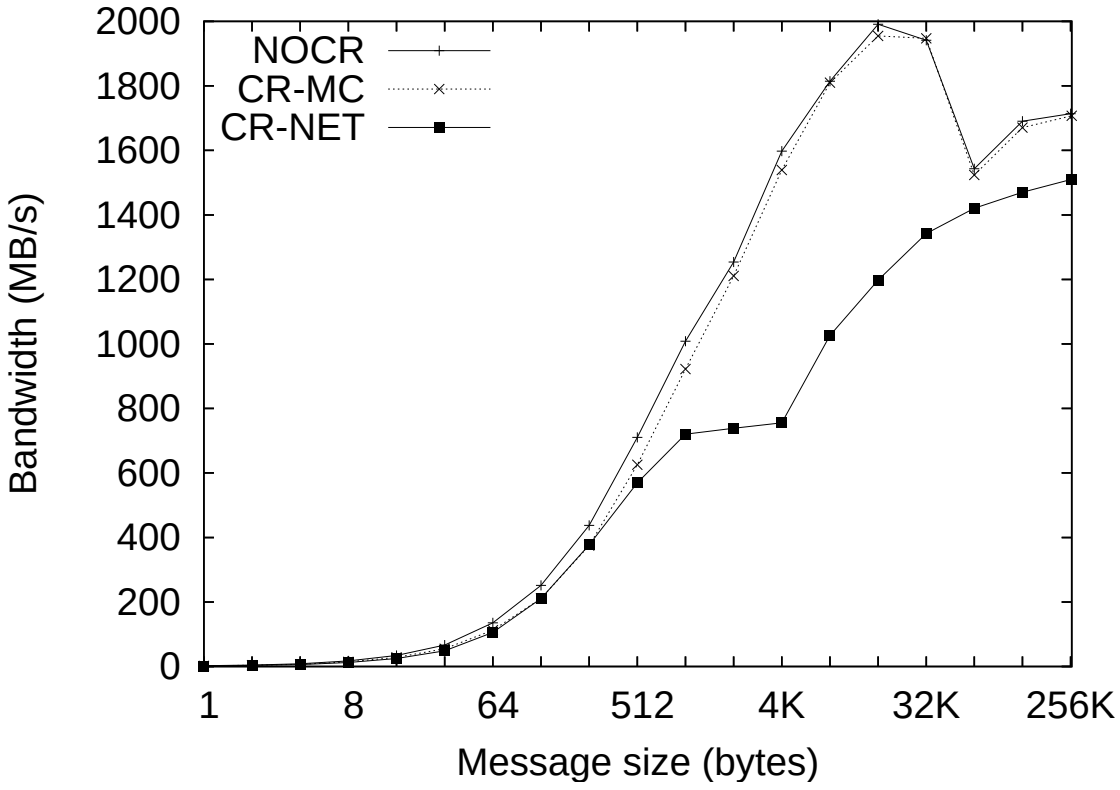


Figure 2.5: Performance of MPI Bandwidth

examine the basic performance of the proposed design. Then we show checkpointing overhead and time breakdown in Section 2.4.4, and process re-distribution effect in Section 2.4.5.

Testbed: We use an Intel Clovertown cluster. Each node is equipped with dual quad-core Xeon processor, i.e. 8 cores per node, running at 2.0GHz. Each node has 4GB main memory. The nodes are connected by Mellanox InfiniBand DDR cards. The operating system is Linux 2.6.18 and the BLCR library used is version 0.6.5. The file system we use is ext3 on top of a local SATA disk.

In this section we will use the following acronyms for different settings:

- NOCR - No Checkpoint/Restart support
- CR-MC - Checkpoint/Restart support with Multi-Channel enabled (design proposed in this paper)
- CR-NET - Checkpoint/Restart support with only the Network channel enabled (design proposed in [30])

It is to be noted that the shared memory channel is available in both NOCR and CR-MC cases (both point-to-point and collectives), but not available in CR-NET.

2.4.1 Impact on Latency and Bandwidth

In this section, we examine the impact of the proposed CR-MC design on MPI intra-node latency and bandwidth. The results are shown in Figures 2.4 and 2.5.

From Figure 2.4 we can see that compared with CR-NET, CR-MC reduces latency significantly. The 4-byte message latency is reduced from 4.55us to 0.76us. This is

because memory copy is much faster than network loopback. Similarly, Figure 2.5 shows that CR-MC increases bandwidth from 1510MB/s to 1954MB/s compared with CR-NET which is 30% improvement.

Comparing with NOCR, we observe that CR-MC adds little overhead, only around 0.1us on latency and almost no overhead on bandwidth. The overhead comes from acquiring and releasing CR locks. This indicates that with CR-MC, users can always run applications with CR support on and decide whether to take checkpoints at run time. If the applications do not take checkpoints, then the performance is not affected. On the other hand, if using CR-NET, the users need to make a decision whether to use CR support at compile time, because with CR-NET the performance may degrade even if no checkpoints are actually taken.

2.4.2 Impact on Collective Operations

In this section, we use IMB[6] to evaluate the performance impact of CR-MC on collective operations. The results of MPI Allreduce and MPI Broadcast on 8 cores and 64 cores are shown in Figures 2.6 and 2.7, respectively. It is to be noted that MPI Allreduce in MVAPICH2 is implemented on top of point-to-point communication for smaller message sizes and uses a special shared memory aware algorithm for larger message sizes. NOCR and CR-MC can exploit the faster intra-node point-to-point communication for MPI Allreduce while CR-NET cannot. MPI Broadcast in MVAPICH2 uses the special shared memory aware algorithm which has optimized performance and is available in NOCR and CR-MC, but not in CR-NET.

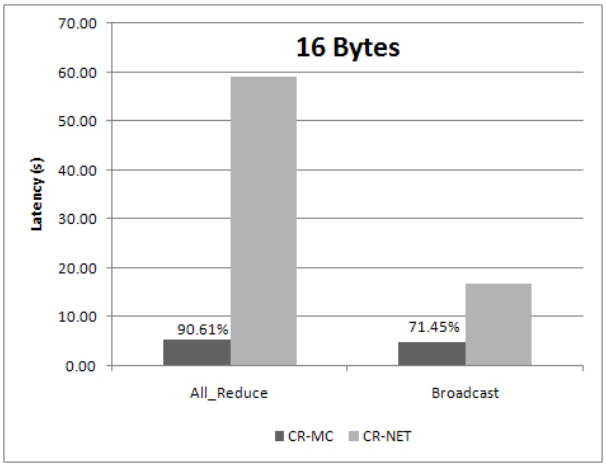
From Figure 2.6 we see that on a single node, CR-MC improves MPI Allreduce latency over CR-NET by 90%, 67% and 32% for 16 byte, 4KB and 16KB messages, respectively. The corresponding improvements for MPI Broadcast are 71%, 17% and 72%. On 64 cores, CR-MC improves MPI Allreduce latency by 85%, 63% and 33% for 16 byte, 4KB and 16KB messages, respectively, and improves MPI Broadcast latency by 76%, 22% and 37% (Figure 2.7). In all cases, CR-MC and NOCR perform comparably.

2.4.3 Impact on Application Performance

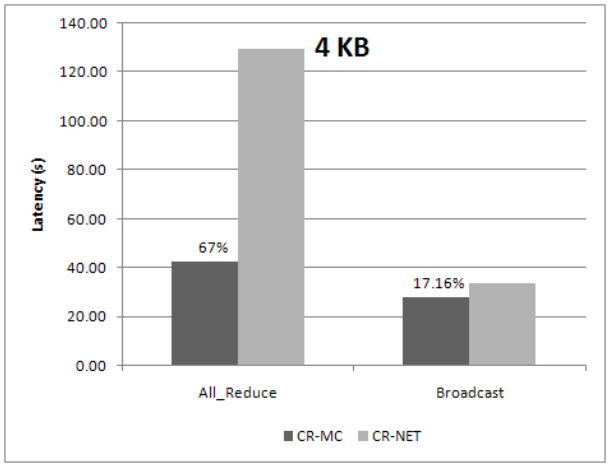
In this section, we evaluate the performance of CR-MC using application level benchmarks, NAS [45], and compare with NOCR and CR-NET. The normalized execution time on 8 cores and 64 cores are shown in Figures 2.8 and 2.9, respectively. From the figures we can see the improvements in latency, bandwidth, and collective operations have been translated into applications. With CR-MC the execution time is reduced by up to 6% compared with CR-NET, which indicates that users can have both CR support and high performance at the same time by using the CR-MC design.

2.4.4 Checkpointing Overhead

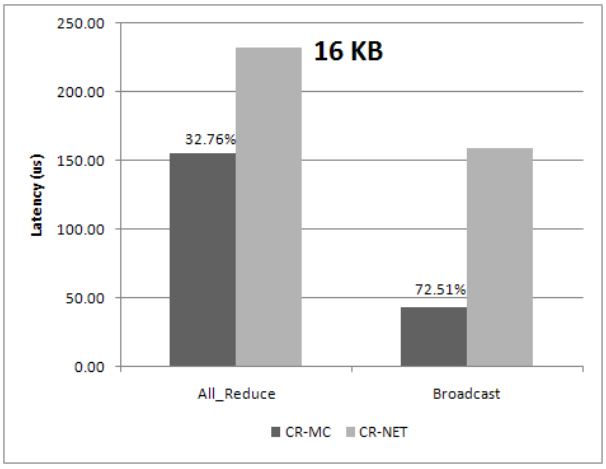
In this section, we use the NAS Benchmark Suite [45] to measure the checkpointing overhead. We run the BT and LU applications (Class C) with one checkpoint and compare the execution time to that without any checkpoints. The result is shown in Figure 2.10. It can be seen that the overhead is around 22% for BT and 16% for LU. The size of the checkpoints is shown in Table 2.1.



(a) 16 Bytes

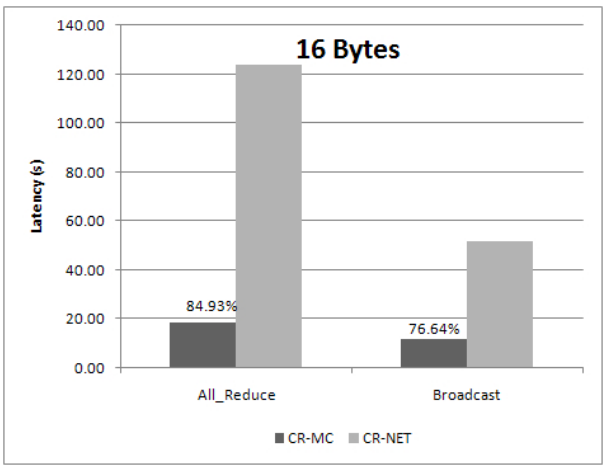


(b) 4KB

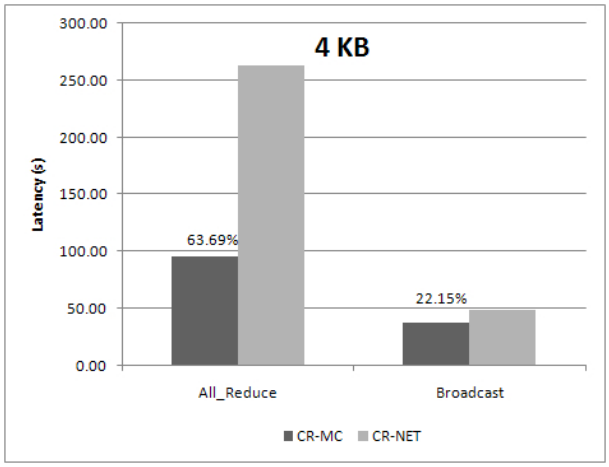


(c) 16KB

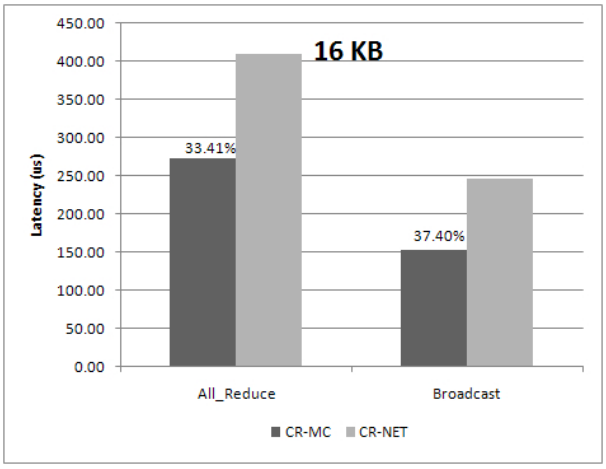
Figure 2.6: Performance of MPI_Allreduce and MPI_Broadcast on 8 Cores (1x8)



(a) 16 Bytes



(b) 4KB



(c) 16KB

Figure 2.7: Performance of MPI_Allreduce and MPI_Broadcast on 64 Cores (8x8)

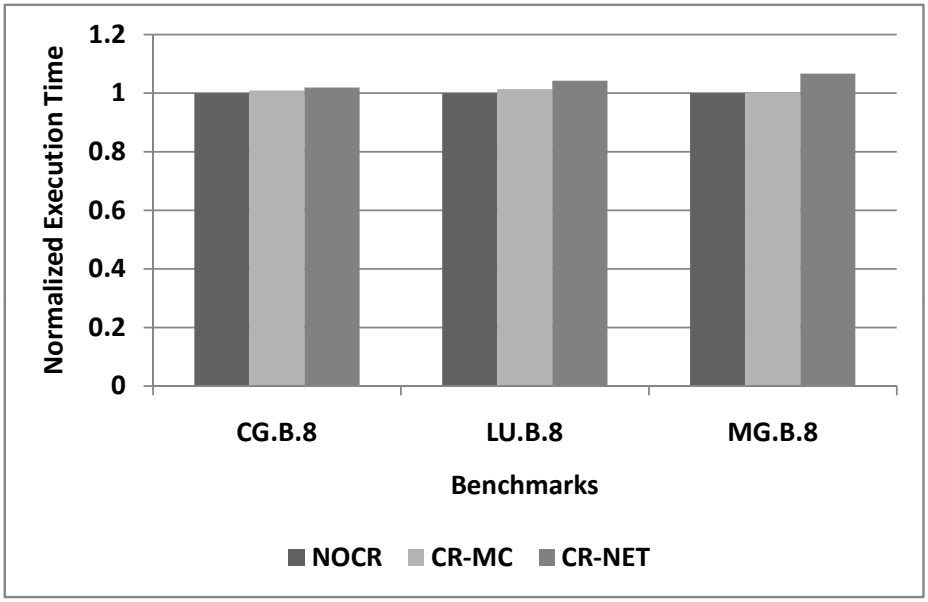


Figure 2.8: Performance Impact of Checkpointing NAS on 8 Cores (1x8)

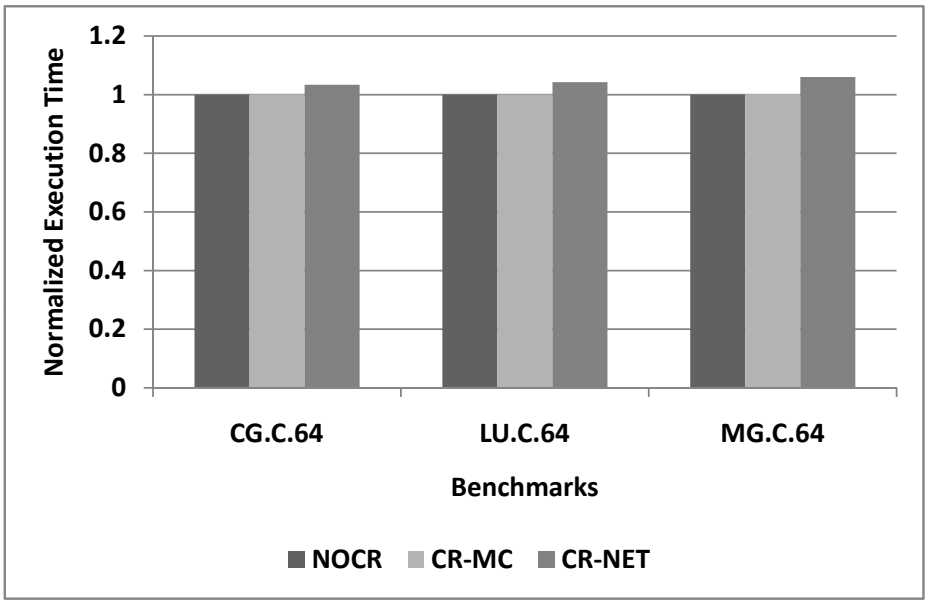


Figure 2.9: Performance Impact of Checkpointing NAS on 64 Cores (8x8)

Table 2.1: NAS Checkpoint File Size

NAS Application	File Size (MB)
BT	2368
LU	1280

The checkpointing time can be broken down into two parts. The coordination time and the file writing time. The coordination time is less than 2% of the overall checkpointing overhead. The file writing time is the dominant factor, which increases with the size of the checkpoint data.

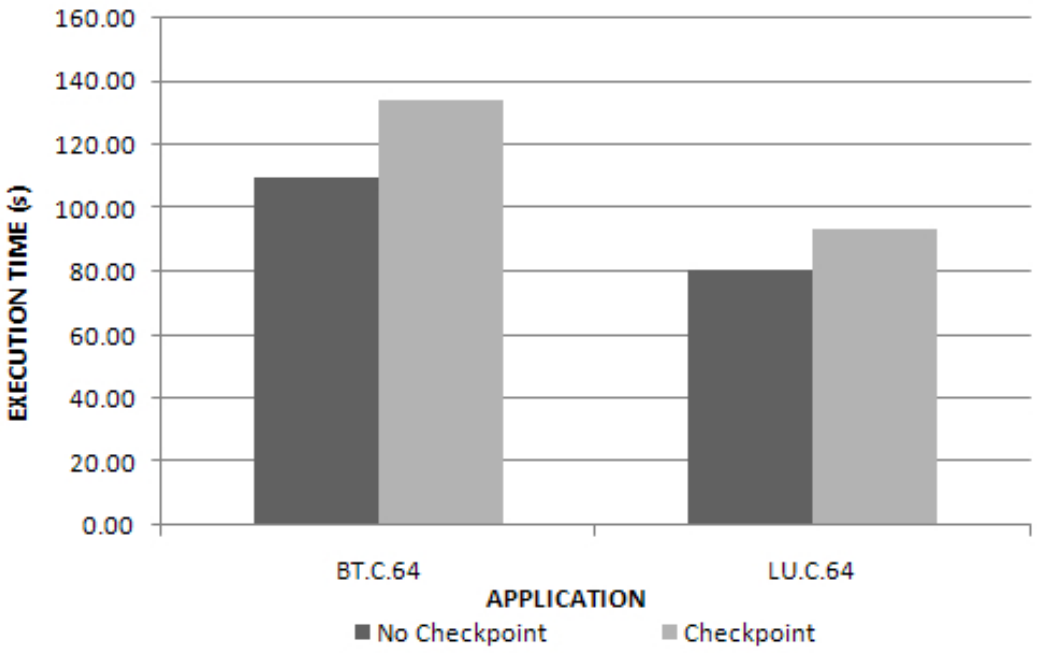


Figure 2.10: Performance Impact of Checkpointing NAS

The file writing time can be reduced by using good parallel file systems (such as Lustre, PVFS2, etc) and high performance storage nodes. Since the focus of this paper is not to optimize the file writing time with parallel file systems, these results are not included here.

2.4.5 Process Re-distribution

In this section, we present the results of process re-distribution. When faults occur and applications need to restart from the previous checkpoint, the processes may need to be re-distributed, e.g. processes previously on different nodes may be re-distributed on the same node or the other way around. This is because the original distribution may not be available after restart. In this set of experiments, we run MPI latency and bandwidth tests with two processes on two different nodes initially, and after 5 iterations we take a checkpoint, and restart the processes on the same node. So starting from the 6th iteration, the processes get re-distributed. The message size is 2KB. The results are shown in Figures 2.11 and 2.12. From the results we see that our design allows process re-distribution and utilizes different channels dynamically.

2.5 Related Work

Fault tolerance in MPI has become a very popular topic in recent times. A lot of research is aimed towards tolerating network faults, including LA-MPI[42], which enables data reliability using multiple network interfaces. Many researchers have proposed multiple schemes to achieve fault tolerance at the MPI application level,

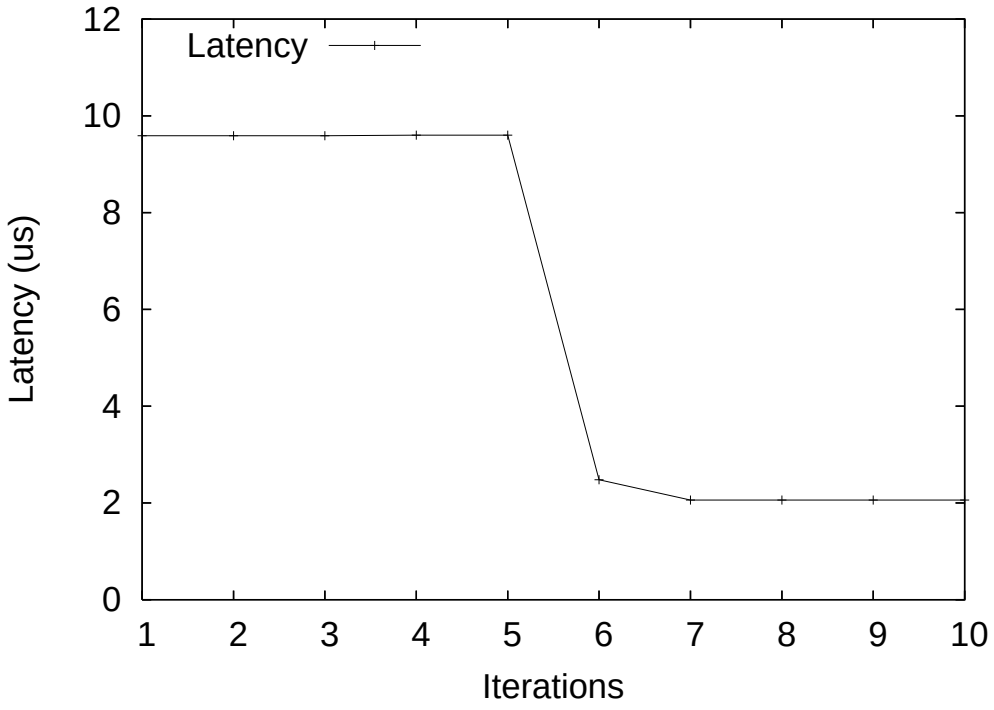


Figure 2.11: Impact of Process Re-distribution on Latency

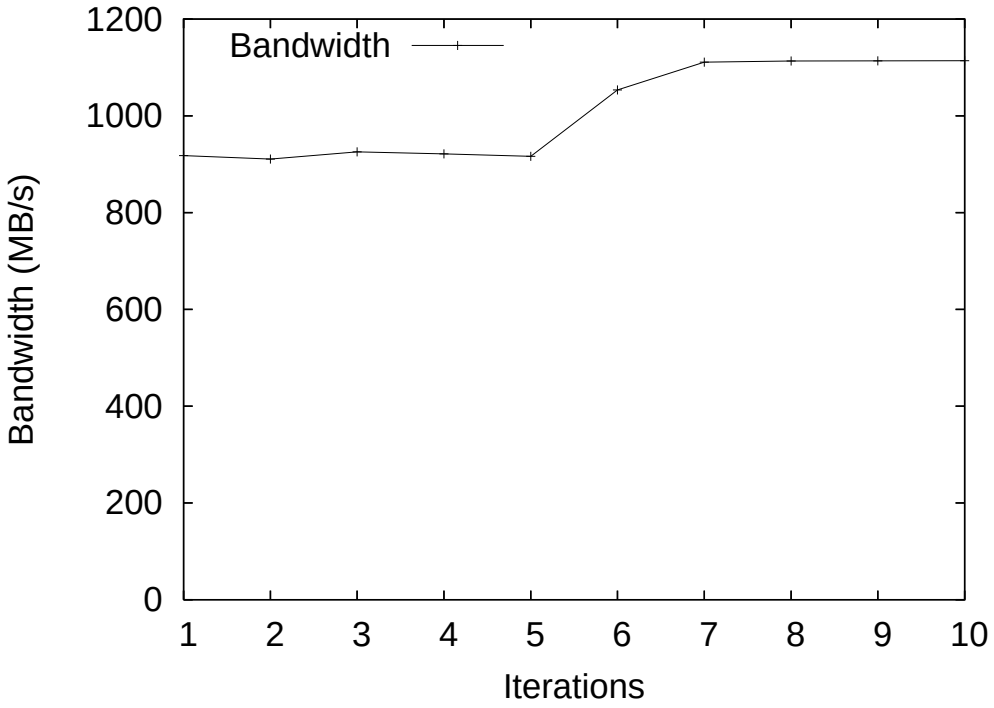


Figure 2.12: Impact of Process Re-distribution on Bandwidth

including work from FT-MPI[27] and from Schulz et al.[41]. From these, it is apparent that a lot of work has been proposed for library based application transparent checkpoint. For example, MPICH-V[17] has developed and evaluated several roll-back recovery protocols. Their work includes uncoordinated checkpointing with message logging[17, 18, 19], and coordinated checkpointing, such as Vcl[35], which is based on the Chandy-Lamport Algorithm[22], and Pcl[23], which is based on coordinated checkpointing. BLCR is a library that provides process-level checkpoint/restart[25] capabilities. Many MPI libraries use BLCR for Checkpointing [40, 30] as well as efficient failure recovery through process migration[44]. In our earlier work [30], we had proposed a framework to checkpoint MPI programs over InfiniBand using a blocking coordinated checkpointing protocol and BLCR.

Most of the work described above assumes a single communication channel. They almost exclusively work on TCP/IP based MPI except our earlier work is on InfiniBand. In reality, however, modern clusters are deployed with multi-core computing nodes connected through high speed interconnects like InfiniBand. Thus, it is important to design a checkpoint framework that can efficiently handle both inter-node and intra-node communication. The work proposed here addresses just that, by proposing a multi-channel enabled checkpoint/restart framework.

2.6 Conclusion

In this chapter, we have presented the design for a framework to checkpoint MPI application that use multiple communication channels. We have also shown that

the framework provides high performance, ease to incorporate new channels into the framework, as well as the flexibility to redistribute processes after restart.

CHAPTER 3

COORDINATED FAULT TOLERANCE FRAMEWORK FOR MPI

High Performance Computing (HPC) Clusters continue to grow to ever increasing proportions. However, since processor speeds no longer double every two years, modern HPC Clusters have begun to scale out, rather than scale up. As a result, there has been an exponential increase in the number of components in the cluster. This has led to considerable deterioration in the MTBF of the cluster. As a result, the support for Fault Tolerance has become an absolute necessity on these clusters. Although considerable research has been conducted with respect to fault tolerance for the individual components in the Hardware / Software stacks, these schemes work in isolation. The lack of a system-wide fault tolerance feature has emerged as one of the big problems on HPC systems.

In this chapter, we present the design for a Coordinated Fault Tolerance Framework for MPI. The framework uses the Fault Tolerance Backplane that has been designed as part of the Co-ordinated Infrastructure for Fault Tolerance Systems (CIFTS). The design enables the MPI software stack to handle faults occurring in the operating environment in a holistic manner.

The rest of the chapter is organized as follows. In section 3.1, we provide a background of the existing state of fault tolerance in MPI. In section 3.2, we provide some information about the Fault Tolerance Backplane. In section 3.3, we provide a brief overview of the design of various FTB components. In section 3.4, we provide a detailed design of the components. In section 3.5, we evaluate our design. Finally, in section 3.6, we present the summary of this chapter.

3.1 Background

Modern High Performance Computing (HPC) Clusters continue to grow to ever increasing proportions. However, performance gains that could be obtained in traditional single core processors by employing schemes such as frequency scaling and instruction pipelining have greatly diminished due to problems in power consumption, heat dissipation and fundamental limitations in exploiting Instruction Level Parallelism. Processor speeds no longer double every 18 - 24 months. As a result, HPC systems have ceased to rely on the speed of a single processing element to achieve the desired performance. They instead tend to exploit the parallelism available in a massive number of moderately fast distributed processing elements which are connected together using a high performance network interconnect.

A quick look at the Top 500 list of high performance machines in the world [15] clearly indicates this trend. The total core count of all the systems on the list as of June 2009 is almost ten times the core count as of June 2004. There has been an order of magnitude increase in the number of cores in the last four years. The Roadrunner System[16], which is currently the world's fastest Supercomputer, has over 128k cores.

Multi-core processors and the availability of commodity high speed interconnects such as InfiniBand[5] has resulted in the trend of using such large clusters. As we usher the era of peta-flop and exa-flop computing, we expect this trend to continue for many more years to come.

However, as these clusters scale out, their Mean Time Between Failures (MTBF) rapidly deteriorates. With this exponential increase in the number of components in the cluster, the MTBF has reduced from days to a couple of hours[32]. Many real world applications that study Molecular Dynamics[3, 2, 12], Finite Element Analysis[9, 38], etc. take anywhere from a few hours to a couple of days to complete their computation. Given that the MTBF of such modern clusters is smaller than the average running time of these applications, multiple failures can be expected during the lifetime of the application. As a result, it has become imperative for these clusters to be equipped with Fault Tolerant capabilities.

Considerable research has been conducted with respect to Fault Tolerance for system software, including the Message Passing Interface (MPI), Network Interconnects, File Systems, resource management infrastructure and applications.

MPI[37] is the *de-facto* standard for Parallel Programming and is widely deployed on most large scale clusters. Most scientific applications [3, 2, 12, 9, 38] as well as Mathematical Libraries used in such applications [13, 14] are written in MPI. Many MPI Libraries provide Fault Tolerance through Checkpointing, Message Logging and redundancy at the network level.

InfiniBand, a high speed low latency cluster interconnect that is commonly used on HPC Clusters, provides features like Automatic Path Migration and Partitioning which can be used for Fault Tolerance.

Although most of the individual hardware and software components within the cluster implement mechanisms to provide some level of fault tolerance, these components work in isolation. They work independently, without sharing information about the faults they encounter. This lack of a system-wide fault tolerance feature has emerged as one of the biggest problems on leadership-class HPC systems.

3.2 The CIFTS Fault Tolerance Backplane

The CIFTS Fault Tolerance Backplane[31] is an asynchronous messaging backplane that provides communication between the various system software components. The Fault Tolerance Backplane (FTB) provides a common infrastructure for the Operating System, Middleware, Libraries and Applications to exchange information related to hardware and software failures in real time. Different components can subscribe to be notified about one or more events of interest from other components, as well as notify other components about the faults it detects.

The FTB physical infrastructure is shown in Figure 3.1. The FTB framework comprises of a set of distributed daemons, called FTB Agents which contain the bulk of the FTB logic and manage most of the book-keeping and event communication throughout the system. FTB agents connect to each other to form a tree-based topology. If an agent loses connectivity during its lifetime, it can reconnect itself to a new parent in the topology tree, making the tree fault tolerant and self-healing.

From the software perspective, the FTB Software Stack consists of three layers, namely, the Client Layer, the Manager Layer, and the Network Layer.

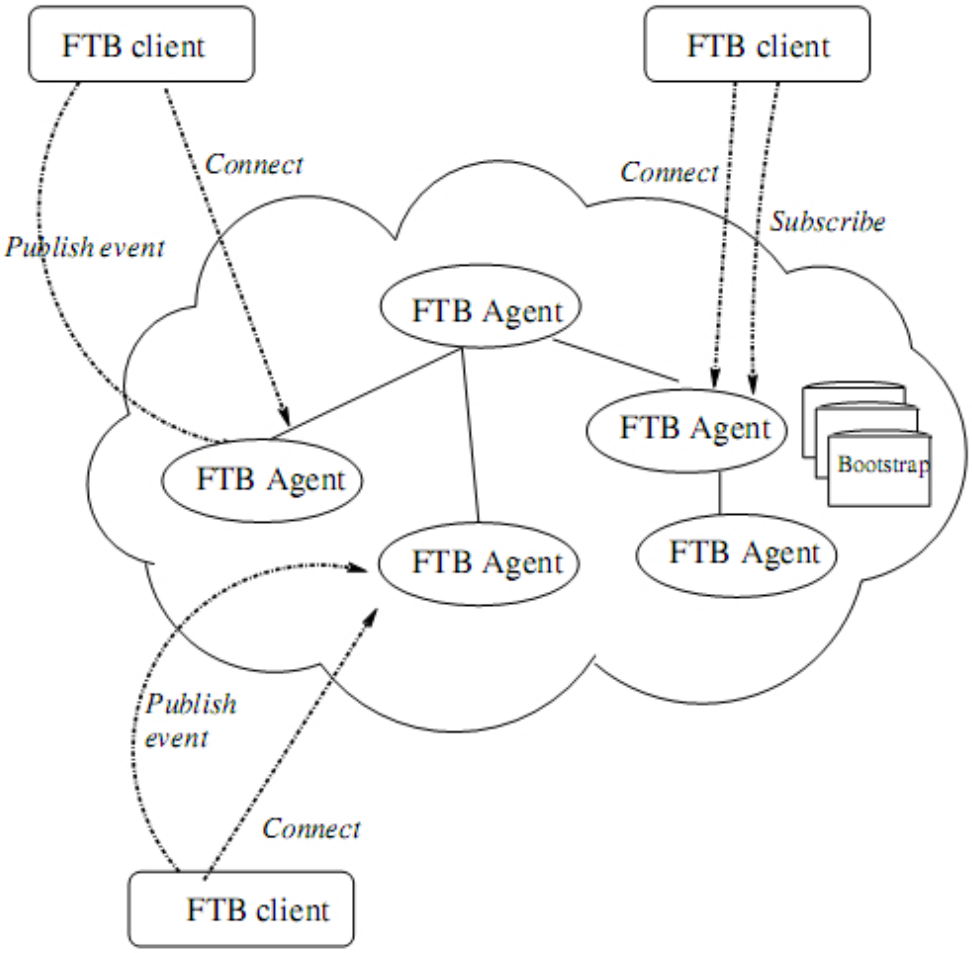


Figure 3.1: FTB Architecture (Courtesy the CIFTS Team)

The Client Layer consists of a set of APIs for the clients to interact with each other. These FTB Client APIs consist of a small set of simple but powerful routines. The "FTB_Connect" API is used by the clients to connect to the FTB framework. Each client has to register itself for a given "Namespace". Once registered, the Namespace to which the client belongs cannot be changed during its lifetime. "FTB_Publish" is used by the clients to advertise a specific event to other clients. All events thrown by the client will belong to the Namespace to which the client had registered. The events can have varying severity such as INFO, WARNING, ERROR or FATAL. Clients which have subscribed to that event will be notified through either synchronously if they poll for events or asynchronously through a registered callback. In addition to the event name, the "FTB_Publish" also allows clients to include a small amount of data as a "payload". This proves to be a useful feature, as will be seen in the future section. "FTB_Subscribe" is used to indicate the Namespace of Events for which the client needs to be notified. The "FTB_Unsubscribe" and "FTB_Disconnect" APIs are used by clients to disassociate from the FTB infrastructure.

The Manager Layer handles the book-keeping and decision making logic. It handles the client subscriptions, subscription mechanisms and event notification criteria. This layer is responsible for event matching and routing events across to other FTB Agents. This layer exposes a set of APIs for the Client Layer to interact with it. The interface is internal to FTB and is not exposed to external clients.

The Network Layer is the lowest layer of the software stack. This layer deals with the sending and receiving of data. The Network Layer is transparent to the upper layers and is designed to support multiple communication protocols such as TCP/IP

and shared-memory communication.

3.3 Design Overview

In this section we present an overview of the design for a Coordinated Fault Tolerance Framework for MPI.

3.3.1 Job Launcher

Modern multi-core HPC environments deploy a Job Launcher to launch multiple instances of the MPI process on the Compute Nodes. ScELA[43], a modern, extensible and high performance Job Launcher used with the MVAPICH2, divides this operation into two phases. In the first phase, the Job Launcher launches a Node Launch Agent (NLA) on each of the Compute Nodes. In the second phase, the NLA on each Compute Node launches the desired number of application instances on its Compute Node. This phase proceeds in parallel on all the Compute Nodes. This process is outlined in Figure 3.2.

In addition to launching the user's MPI application, the Job Launcher is also involved during the Checkpoint / Restart phase. When the MPI Application is requested to be checkpointed, either by the user, or by the Application itself, the Job Launcher is responsible for propagating this request to the MPI Library. The Job Launcher is similarly involved during the Restart as well. The detailed design has been described in [34] and [30].

In this work, we propose the addition of a "FTB Manager" to the Job Launcher. This component is responsible for sending and receiving fault related information from

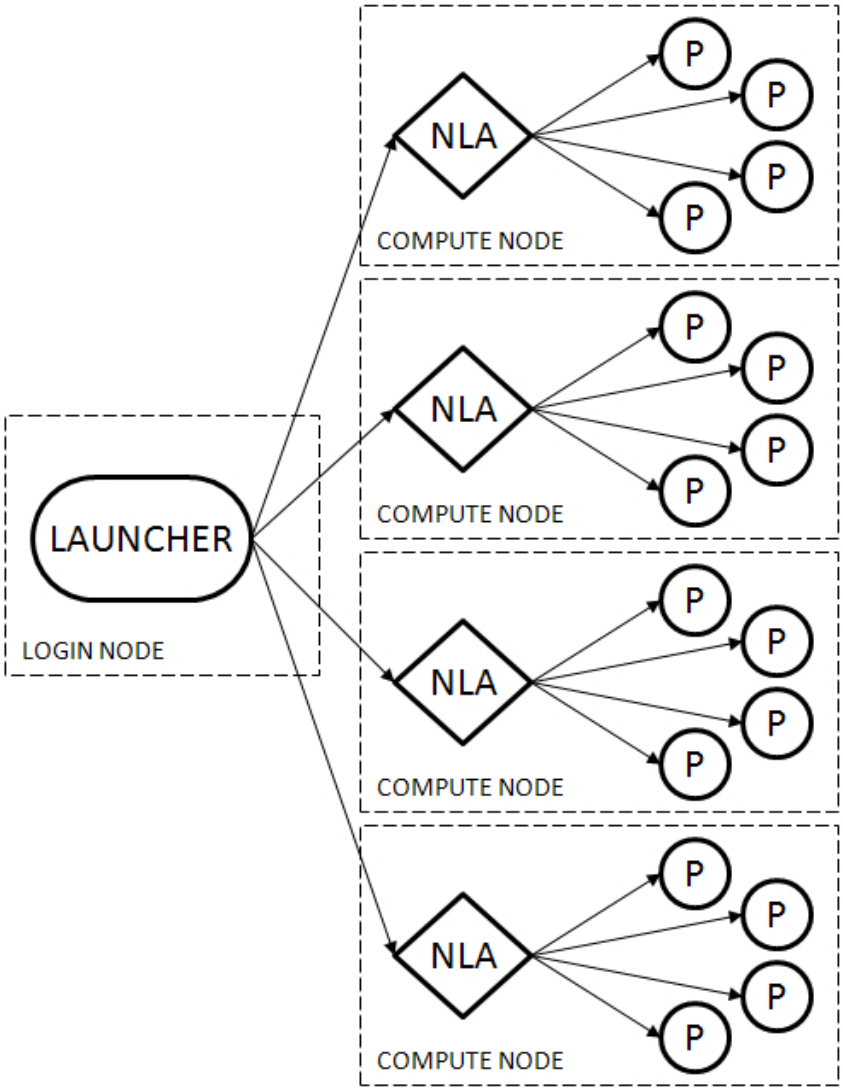


Figure 3.2: ScELA - Launch Process

the rest of the system. This would include user-initiated Checkpoint and Restart requests, as well as information pertaining to the health of the compute nodes. Based on the information received, the FTB Manager would take necessary actions, including, triggering a checkpoint, migrating processes away from a failing node, etc.

3.3.2 MPI Library

The MPI Library as described in [34] consists of a "CR Manager" which waits for Checkpoint / Restart requests from the Job Launcher. However, the existing framework uses a closed scheme for this communication. As a result, Job Launchers that are not CR-aware cannot be used on such systems. Some amount of effort is necessary to enable this functionality.

In this work, we propose the inclusion of an "FTB Layer" in the "CR Manager" to use FTB for the communication. This "FTB Layer" would be responsible for communicating fault information from the Job Launcher to the MPI library, and by the MPI library to send notification to the Job Launcher about failures during the Checkpoint or Restart phases.

3.3.3 InfiniBand Component

The InfiniBand Architecture Specification[5] defines APIs as part of the Verbs Interface that can be used by system and application software to access information about Asynchronous Events from the InfiniBand Adapter. The InfiniBand Verbs library provides facilities wherein the application can register an "Event Handler" which would be invoked to signal an Event.

The FTB-IB component[4], designed as part of this work, uses the FTB Infrastructure to notify other FTB enabled components about failures in the InfiniBand network. FTB-IB uses the Asynchronous Event Handler provided by the native InfiniBand Verbs library to receive information about faults in the InfiniBand network.

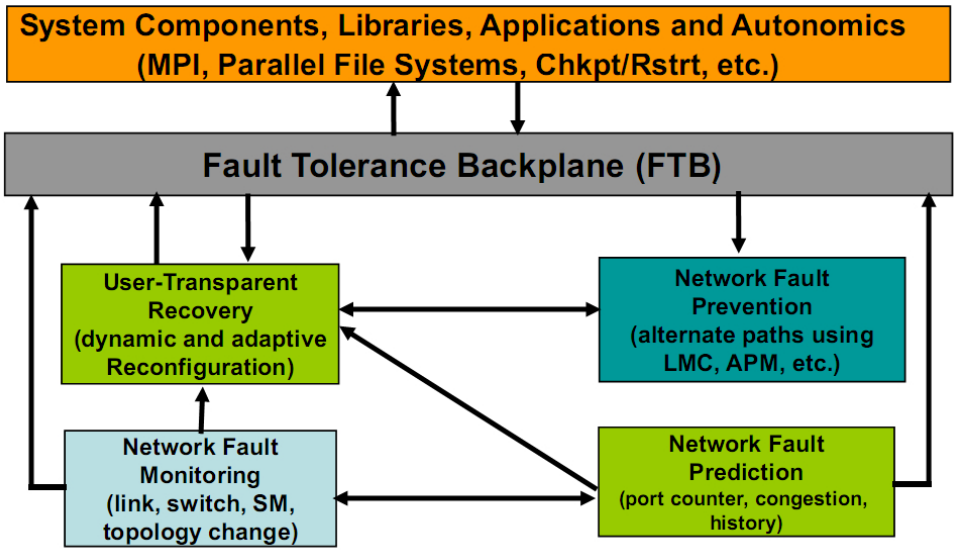


Figure 3.3: FTB-IB - Current State and Future Plans

Figure 3.3 shows the overall architecture of FTB-IB. The Network Fault Monitoring block is responsible for monitoring the InfiniBand fabric for changes in state of the point-to-point links. The Network Fault Prediction block is responsible for using information exported by statistics counters to record network congestion events, in an effort to proactively predict network faults. The Network Fault Prevention block uses techniques such as Automatic Path Migration, provided by InfiniBand to workaround

faults. These three blocks work with each other to achieve User-Transparent Recovery. Network Fault Prevention is implemented as a layer in the MVAPICH2 [11] MPI Library. The current version of FTB-IB implements Network Fault Monitoring.

Appendix A contains a list of supported events that are exposed by FTB-IB to the FTB Framework.

3.3.4 IPMI Component

The Intelligent Platform Management Interface (IPMI) [7] defines a set of common interfaces to a computer system which can be used to monitor system health. IPMI consists of a main controller called the Baseboard Management Controller (BMC) and other management controllers distributed among different system modules that are referred to as Satellite Controllers (SC). The BMC connects to SCs within the same chassis through the Intelligent Platform Management Bus/Bridge (IPMB) and to other SCs or BMCs on another chassis through the Intelligent Chassis Management Bus/Bridge (ICMB). The overall architecture of IPMI is shown in Figure 3.4.

Amongst other pieces of information, IPMI maintains a Sensor Data Records (SDR) repository which provides the read outs from individual sensors present on the system, including, sensors for voltage, temperature and fan speed. System administrators can use IPMI calls to query these values.

In this work, we design a FTB-IPMI component, which uses the IPMI interface to gather information about the system's health and notifies other FTB enabled components about these events using the FTB Infrastructure.

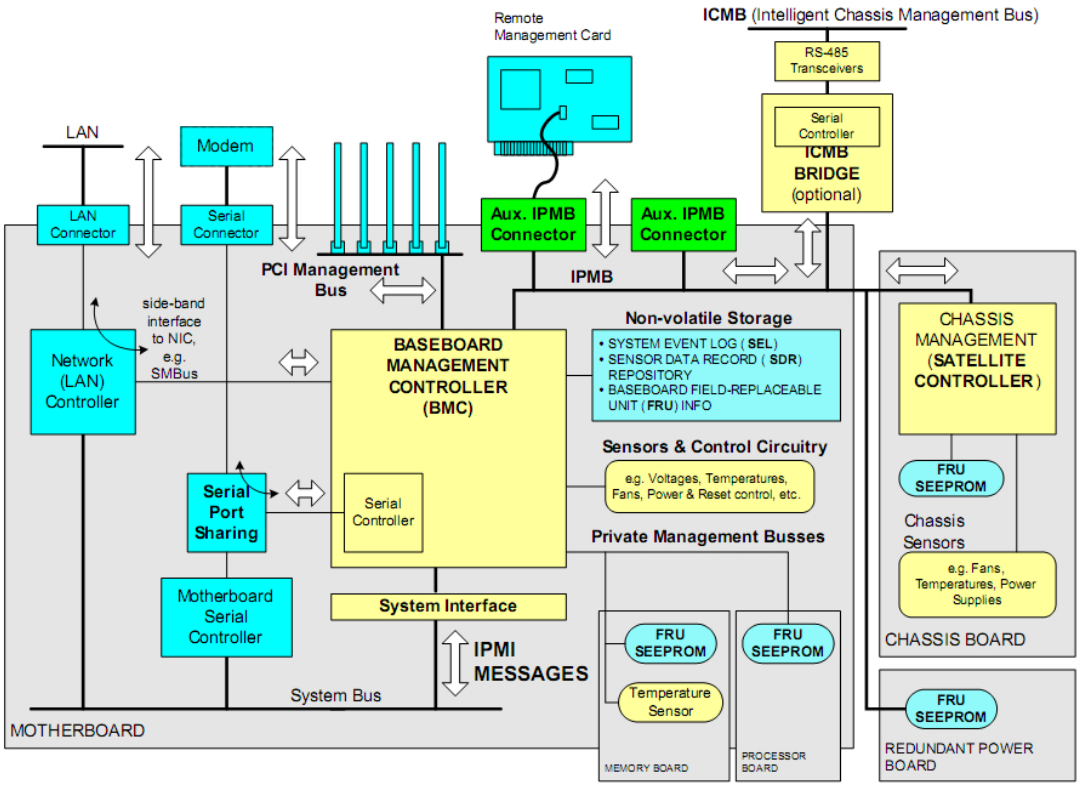


Figure 3.4: IPMI System Architecture (Courtesy, IPMI Specification)

Appendix B contains a list of supported events that are exposed by FTB-IPMI to the FTB Framework.

3.4 Detailed Design

The architecture proposed in this work is as shown in Figure 3.5. The FTB Agent is active on every node, including Login and Compute Nodes, as depicted in the Figure. These agents connect with each other to form a reliable communication framework, as described in Section 3.2. The other entities on the node, such as the Job Launcher, MPI Processes and FTB-IPMI use the FTB API to communicate with each other, through the FTB Agents. The solid lines indicate the actual communication path, while the dashed lines depict the logical communication between the different components. The arrow heads indicate the direction of information flow.

The Job Launcher, NLA, MPI Library and FTB-IPMI register themselves under the "FTB.JL", "FTB.NLA", "FTB.MPI" and "FTB.IPMI" Namespaces, respectively. The Job Launcher subscribes to the "FTB.NLA", "FTB.MPI" and "FTB.IPMI" Namespaces. The NLA and MPI Library subscribe to the "FTB.JL" namespace. The directed communication graph is shown in Figure 3.6.

We now proceed to discuss the protocols used to Checkpoint, Restart and Migrate MPI Processes using FTB.

3.4.1 Checkpoint / Restart Protocol

When a Checkpoint needs to be taken, the "FTB_CHECKPOINT" message is published by the Job Launcher. The MPI Library on all processes receives this

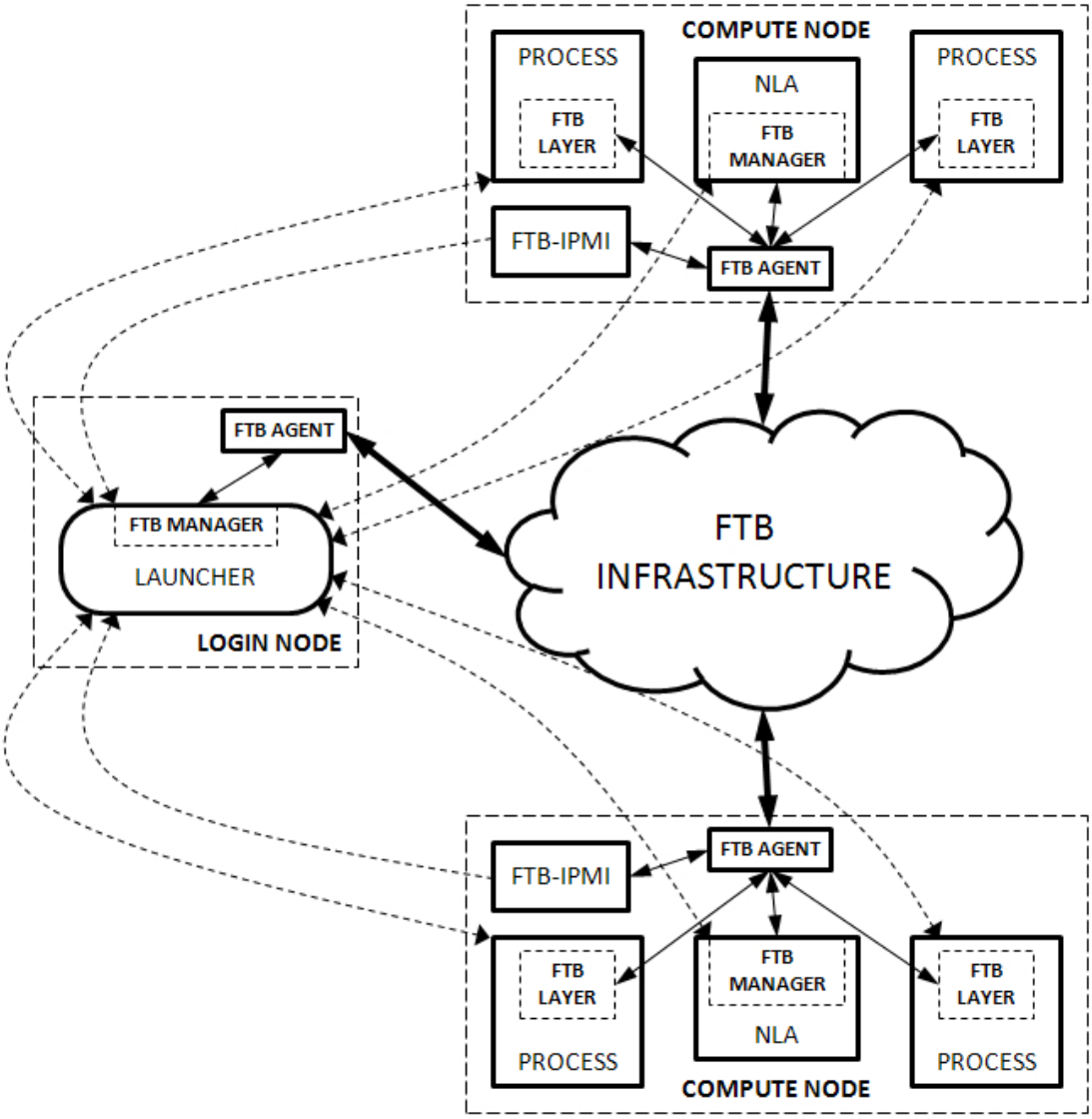


Figure 3.5: Coordinated Fault Tolerance Framework for MPI

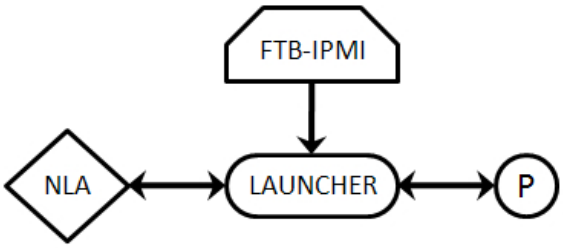


Figure 3.6: Directed Communication Graph of FTB Components

message. On receiving the message, the MPI Libraries take a checkpoint as described in Section 2.3. Once the MPI Library completes taking the Checkpoint, it publishes the "FTB_CHECKPOINT_DONE" message. When the Job Launcher receives this message from the MPI Library, it checkpoints itself. The checkpoint images of the MPI Library and Job Launcher collectively constitute the MPI Job's Checkpoint. The NLA is not involved in the Checkpoint process. The process is depicted in Figure 3.7.

When an MPI Job needs to be Restarted from a previous Checkpoint, the Job Launcher is first restarted from its checkpoint image. On restart, the Job Launcher launches the NLAs on the specified nodes, which in turn restart the MPI Processes from the checkpoint images. FTB is not involved during the Restart process.

3.4.2 Process Migration Protocol

When an MPI Job is executed, a list of "Hot Spare Nodes" is specified to the Job Launcher. During the first phase of the Job Launch process, as described in Section 3.3.1, the Job Launcher launches the NLAs on the Hot Spare Nodes, in addition

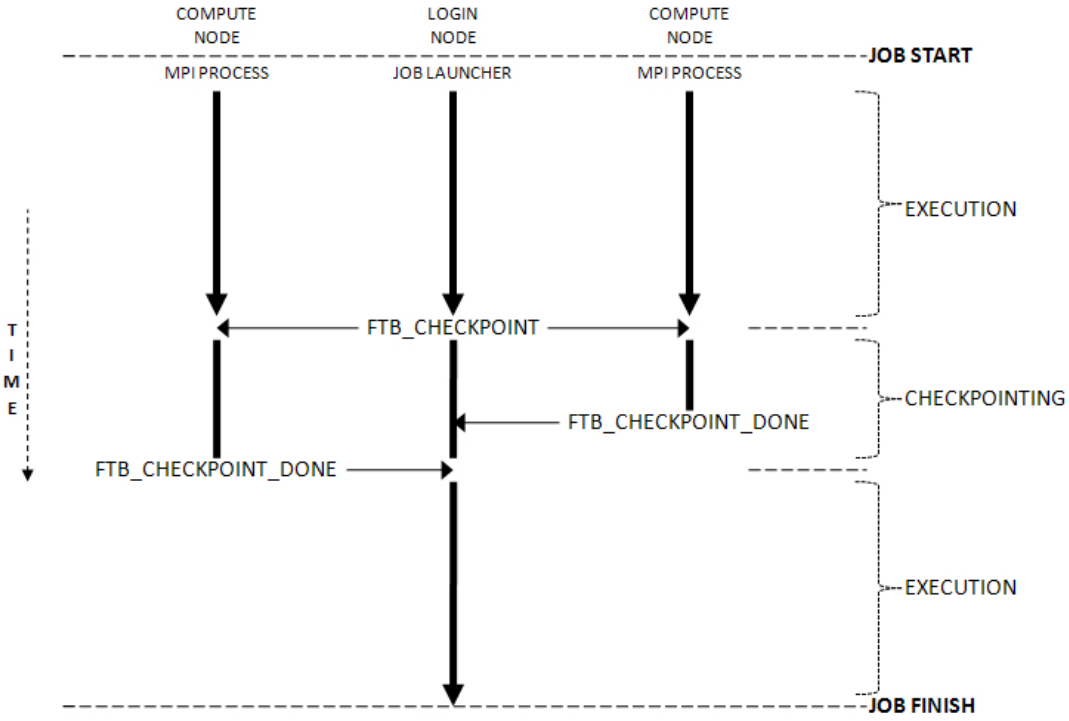


Figure 3.7: FTB based Checkpoint

to launching them on the Primary Compute Nodes. The NLAs on all the Primary Compute Nodes are in the "MIGRATION_READY" state, while those on the Hot Spare Nodes are in the "MIGRATION_SPARE" state.

When processes need to be migrated away from a Primary Compute Node, the Job Launcher publishes the "FTB_MIGRATE" message, along with the hostname of the Migration Source Node as the Payload. This message is received by all the NLAs and all MPI Processes. On receiving, this message, the NLA on the Migration Source Node transitions its state from "MIGRATION_READY" to "MIGRATION_ARM" preparing itself for the Migration.

At the same time, all the MPI Processes, which also receive the "FTB_MIGRATE" message, suspend all their MPI communication activity and tear down their communication end-points. MPI Processes that are not on the Migration Source Node enter a "Migration Barrier". Like any other barrier, processes which enter the Barrier leave only when all the processes are in the barrier. MPI Processes that are on the Migration Source Node take a checkpoint using BLCR and exit without entering the Migration Barrier. As a result, processes on the other nodes continue to wait in the barrier.

The NLA on the Migration Source Node, which is in the "MIGRATION_ARM" state, waits for all its child MPI Processes to exit after taking a checkpoint. Once the MPI Processes exit, the NLA publishes the "FTB_MIGRATE_PIC" message, to indicate the completion of Phase I of the Migration Process to the Job Launcher. After doing so, the NLA transitions to the "MIGRATION_INACTIVE" state, which indicates that the NLA's node is no longer active.

On receiving the "FTB_MIGRATE_PIC" message, the Job Launcher selects a node from the list of Hot Spare Nodes as the Migration Target Node. Once the node is selected, the Job Launcher copies the checkpoint images from the Migration Source Node to the Migration Target Node. Once the images are copied, it sends the "FTB_MIGRATE_IPT" message to request the initiation of Phase II of the Migration Process. The Payload of this message contains the hostname of the Migration Target Node and the list of ranks of the MPI Job that were running on the Migration Source Node. Using this information, the NLA on the Migration Target Node restarts the MPI Processes from their checkpoint images. The NLA on the Migration Target Node transitions its state from "MIGRATION_SPARE" to "MIGRATION_READY", to indicate that it is now active.

The MPI Processes that have been restarted on the Migration Target Node enter the Migration Barrier. At this point of time, all processes are in the barrier. As a result, all the process are now free to leave the barrier. Once out of the barrier, all the processes re-establish their communication end-points and resume their MPI communication activity.

At this point of time, the Process Migration cycle is complete and is ready for the next migration request. The entire process is depicted in Figure 3.8.

3.4.3 Checkpoint / Migration Trigger

Sections 3.4.1 and 3.4.2 discuss the Checkpoint, Restart and Migrate mechanisms once the Job Launcher receives the request to do so. This section deals with the mechanisms to deliver the request. As has been discussed in Section 3.1, the ability

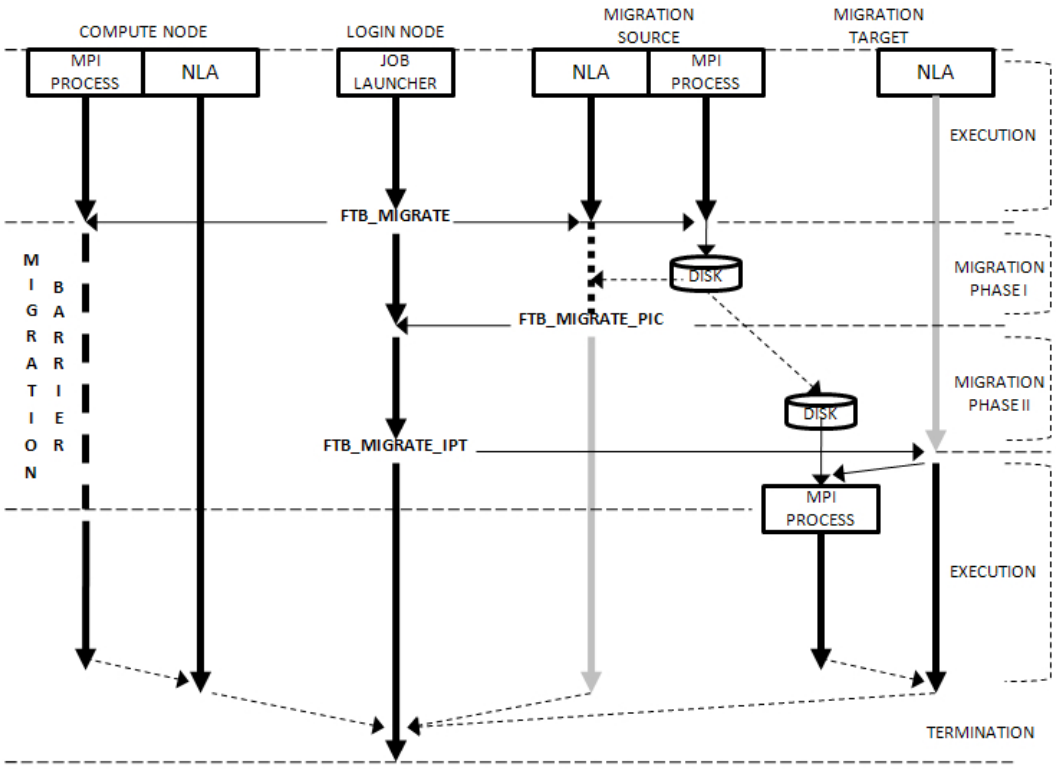


Figure 3.8: FTB based Migration

to receive fault related information from a variety of sources is of utmost importance. In this section, we consider the example of the FTB-IPMI Daemon.

The FTB-IPMI daemon uses the IPMI subsystem to query the Sensor Data Repository for events of interest. And excerpt of the information obtained from IPMI is shown in Tables 3.1 through 3.4.

Table 3.1: IPMI Sensor Data Repository - BaseBoard Voltages

Record Name	Record Locator	Status	EntityID. Instance	Value
BB +1.2V Vtt	10h	ok	7.1	1.20 Volts
BB +1.5V AUX	12h	ok	7.1	1.49 Volts
BB +1.5V	13h	ok	7.1	1.47 Volts
BB +1.8V	14h	ok	7.1	1.80 Volts
BB +3.3V	15h	ok	7.1	3.37 Volts
BB +3.3V STB	16h	ok	7.1	3.32 Volts
BB +1.5V ESB	17h	ok	7.1	1.48 Volts
BB +5V	18h	ok	7.1	5.07 Volts
BB +12V AUX	1Ah	ok	7.1	12.03 Volts
BB 0.9V	1Bh	ok	7.1	0.90 Volts

Table 3.2: IPMI Sensor Data Repository - Thermal Data

Record Name	Record Locator	Status	EntityID. Instance	Value
Baseboard Temp	30h	ok	7.1	29 degrees C
P1 Therm Margin	99h	ok	3.1	-40 degrees C
P2 Therm Margin	9Bh	ok	3.2	-41 degrees C

Table 3.3: IPMI Sensor Data Repository - Fan Speed

Record Name	Record Locator	Status	EntityID. Instance	Value
CPU 1 FAN	50h	ok	29.1	2244 RPM
CPU 2 FAN	51h	ok	29.3	2046 RPM
SYS FAN 2 TACH	53h	ok	29.9	17850 RPM
SYS FAN 3 TACH	54h	ok	29.4	4200 RPM
SYS FAN 4 TACH	55h	ok	29.2	4270 RPM

Table 3.4: IPMI Sensor Data Repository - Memory Status

Record Name	Record Locator	Status	EntityID. Instance	Value
DIMM A1	E0h	ok	32.1	Device Installed
DIMM A2	E1h	ok	32.2	
DIMM B1	E2h	ok	32.3	Device Installed
DIMM B2	E3h	ok	32.4	
DIMM C1	E4h	ok	32.5	Device Installed
DIMM C2	E5h	ok	32.6	
DIMM D1	E6h	ok	32.7	Device Installed
DIMM D2	E7h	ok	32.8	
MemA Error	ECh	ok	7.1	
MemB Error	EDh	ok	7.2	
MemC Error	EEh	ok	7.3	
MemD Error	EFh	ok	7.4	

Table 3.1 contains information pertaining to the BaseBoard's Voltages. For example, the "BB +1.5V" record indicates that the voltage on the +1.5V line is currently 1.47V. The Status shows "ok" since it is within the acceptable tolerance. Table 3.2 provides Thermal Data. The "Baseboard Temp" record indicates that the temperature of the system's BaseBoard is 29 degrees Celsius. The next two records indicate the two CPU sockets' Thermal Margin. For instance, the CPU on the first socket can withstand a 40 degree Celsius increase in temperature. Table 3.3 lists the status and speed of the various Cooling Fans on the system. Table 3.4 displays information about the available DIMMs. It shows that Banks A1, B1, C1 and D1 have memory modules installed and that no errors have been detected on any of them. Based on all this information, FTB-IPMI can make an informed decision about the status of the system.

Consider the scenario in which the Fan on one of the Sockets has failed. In this case, there would be a rapid deterioration in that Socket's Thermal Margin. FTB-IPMI will observe this and publish the "FTB_IPMI_FANFAIL" event. On receiving this event, the Job Launcher would immediately migrate MPI Processes away from the failing node.

Consider a scenario in which systems on a certain rack are not sufficiently cooled due to improper air circulation in the air conditioning ducts in their vicinity. This would result in those servers over-heating. In this case, FTB-IPMI on those nodes would detect this thermal deterioration and publish the "FTB_IPMI_OVERTEMP" event. The Job Launcher could then try to migrate processes away from those nodes and mark those nodes as unusable. Once the problem is fixed, the temperature on those nodes would return to normal, in which case, FTB-IPMI on those nodes would

publish the "FTB_IPMI_NORMAL" event. The Job Launcher could then mark those nodes as usable again.

Consider another scenario in which the Data Center, housing the HPC Systems, experiences a failure in the Air Conditioning Unit. This would translate to a gradual deterioration in the Thermal Margin of all CPU Sockets on all systems. In this case, FTB-IPMI on all nodes would publish the "FTB_IPMI_OVERTEMP" event. The Job Launcher would observe this trend and see an impending failure across the entire cluster. It would take a checkpoint of the entire MPI Job to be able to recover at a later point in time.

3.5 Performance Evaluation

In this section, we evaluate the different aspects of the FTB based design. We use MVAPICH2 [11], a High Performance MPI Library for InfiniBand, to evaluate our designs. The hardware setup consists of an Intel Clovertown cluster. Every node in the cluster is equipped with 2 Sockets, each having a Quad-Core Intel E5345 Xeon Processor, and has 4GB of Main Memory. The nodes are connected using Mellanox MT25418 InfiniBand DDR adapters. The nodes run RHEL Server 5.2. All performance numbers have been obtained using BLCR 0.8.0, and FTB 0.6.0.

3.5.1 Code Complexity

A major advantage of using FTB to propagate fault information across the cluster lies in the fact that, components which publish the fault information do not have to be aware of components that have subscribed to that information. The FTB

infrastructure handles the message delivery to all intended targets. As a result, the messaging layer of all components is greatly simplified. In this section, we evaluate this benefit of using the Fault Tolerance Backplane to handle the communication of fault information, in the implementation of the Checkpoint / Restart feature, in terms of the code complexity.

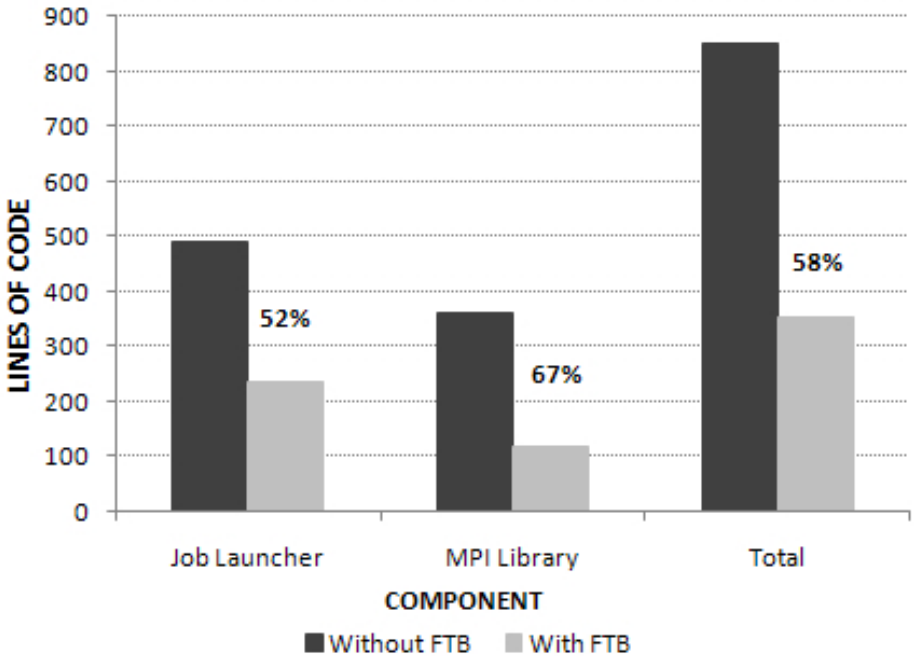


Figure 3.9: FTB Code Complexity

From Figure 3.9, it can be observed that using the Fault Tolerance Backplane to handle the communication of the control messages between the Job Launcher and MPI Library significantly reduces the code complexity. The size of the messaging layer in the Job Launcher reduces from 488 lines to 235 lines, a 52% reduction in code complexity. Similarly, the messaging layer in the MPI Library reduces from 361

lines to 119 lines, a 67% reduction in code complexity. The overall code complexity is reduced by 58%, from 849 lines to 354 lines. This reduction is primarily due to the simple but powerful set of APIs that FTB provides.

3.5.2 FTB Agent CPU Utilization

FTB Agents, running on the Login and Compute Nodes, are responsible for sending messages from the event source to the target entities. It is of paramount importance for the agents to consume as little CPU time as possible. In this section, we measure the CPU Utilization of the FTB Agents at various loads.

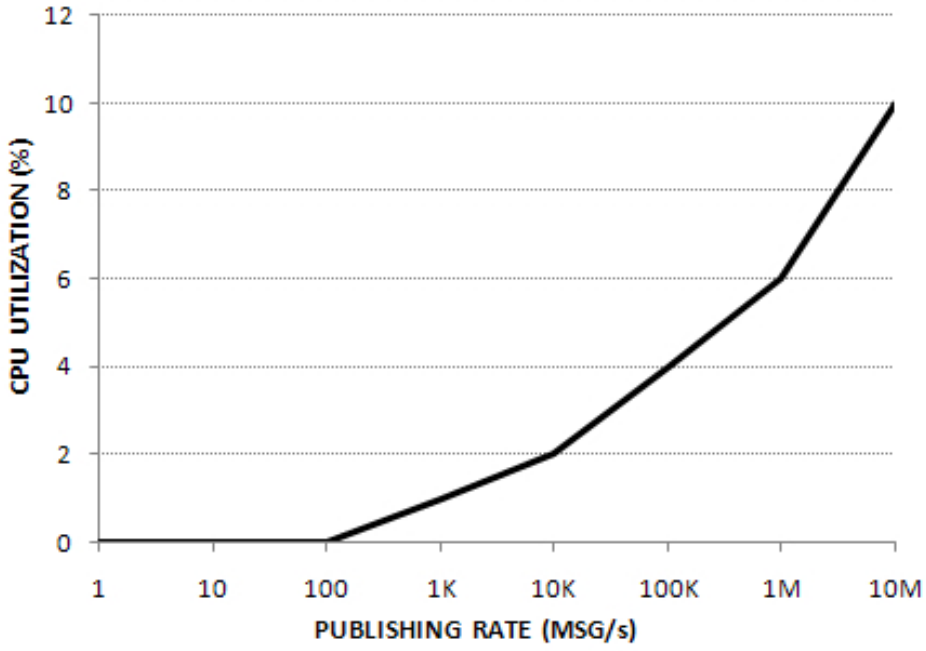


Figure 3.10: FTB Agent CPU Utilization

To measure the CPU Utilization, we conduct an experiment by starting the FTB Agents on thirty three Nodes. On the first Node, we start "FTB Notifier", an FTB component which receives all FTB events. On the remaining thirty two nodes, we launch "FTB Thrower", a component to publish FTB events at a specified rate. The FTB messages are routed by the agents, through the tree topology and eventually reach the first node. The agent on the first node performs the event matching. Figure 3.10 shows the CPU Utilization of the FTB Agent on the first node for varying message rates. It can be observed that the CPU Utilization is significant only when the Publish Rates are larger than 100K messages per second. Publish rates would never be this high in any real HPC environment. Hence, the FTB Agent's CPU Utilization is reasonable.

3.5.3 Impact of FTB-IPMI on Application Performance

As discussed in Section 3.4.3, FTB-IPMI can be used to trigger a Checkpoint or Migration, based on the health of the nodes. As a result, an instance of the FTB-IPMI daemon will be running on every compute node. Since the MPI application itself would also be executing on the node, it is important for FTB-IPMI to have as little an impact on the application as possible.

To measure the effect, of FTB-IPMI on MPI Applications, we run applications from the NAS Parallel Benchmark Suite[45] on a set of compute nodes, while the FTB-IPMI daemons monitor the nodes' health. FTB-IPMI queries the Sensor Data Repository once every 10 seconds. Figure 3.11 shows the impact of this operation on the NAS Benchmarks.

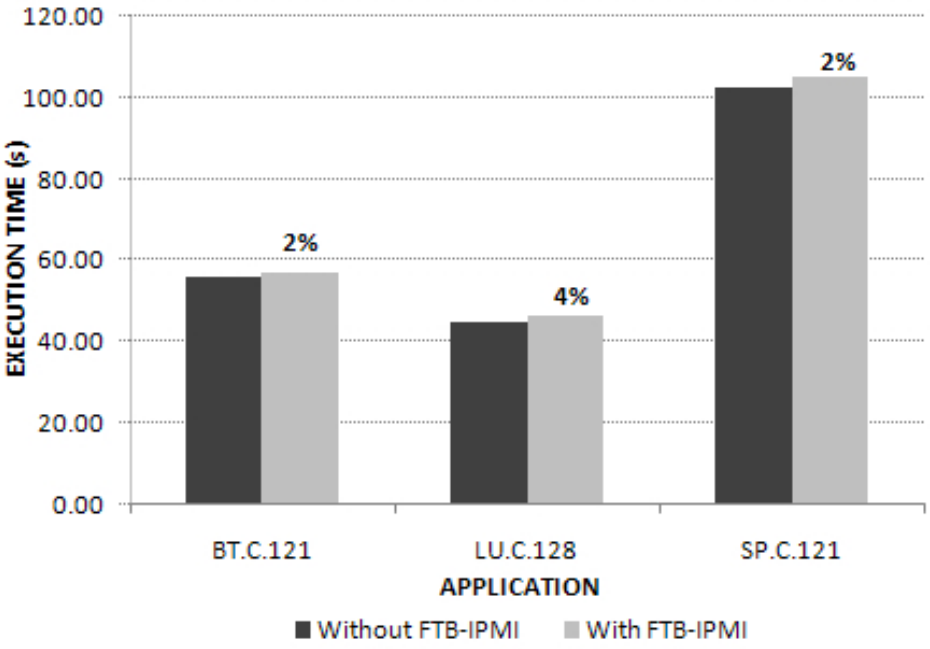


Figure 3.11: Impact of FTB-IPMI on Application Performance

It can be observed that the Execution Time of the Class C applications goes up marginally by 2% to 4%. This is due to the fact that FTB-IPMI makes an IPMI request to the IPMI Driver. The driver spawns a kernel thread, which sends the request to the IPMI hardware and polls for the request's completion. The polling operation consumes CPU cycles which degrades the Application's performance.

The application's performance degradation can be decreased by reducing the frequency with which FTB-IPMI queries the Sensor Data Repository. This would however negatively impact the response time of the Fault Tolerance Framework in the event of an impending failure. The performance degradation can also be decreased by using an event-driven mechanism provided by certain IPMI Management Utilities[8]. By doing so, FTB-IPMI would be asynchronously notified about any event of interest, rather than having to poll the IPMI hardware, which consumes CPU cycles.

3.5.4 Impact on the File System

In this section, we evaluate the impact of Process Migration on the File System, compared to a full fledged checkpoint. We use the 64 process Class C, LU, CG and BT applications, and the 128 process Class D LU, CG and BT applications from the NAS Parallel Benchmark Suite[45] for this study. Figure 3.12 compares the sizes of the images written to disk in the case of a Checkpoint and Migration.

It is observed that the Process Migration scheme consumes much lesser disk resources than a full fledged checkpoint. As a result, the overall migration sequence complete more quickly than a checkpoint and substantially reduces the stress on the file system. Furthermore, by enhancing the checkpointing library, the MPI processes

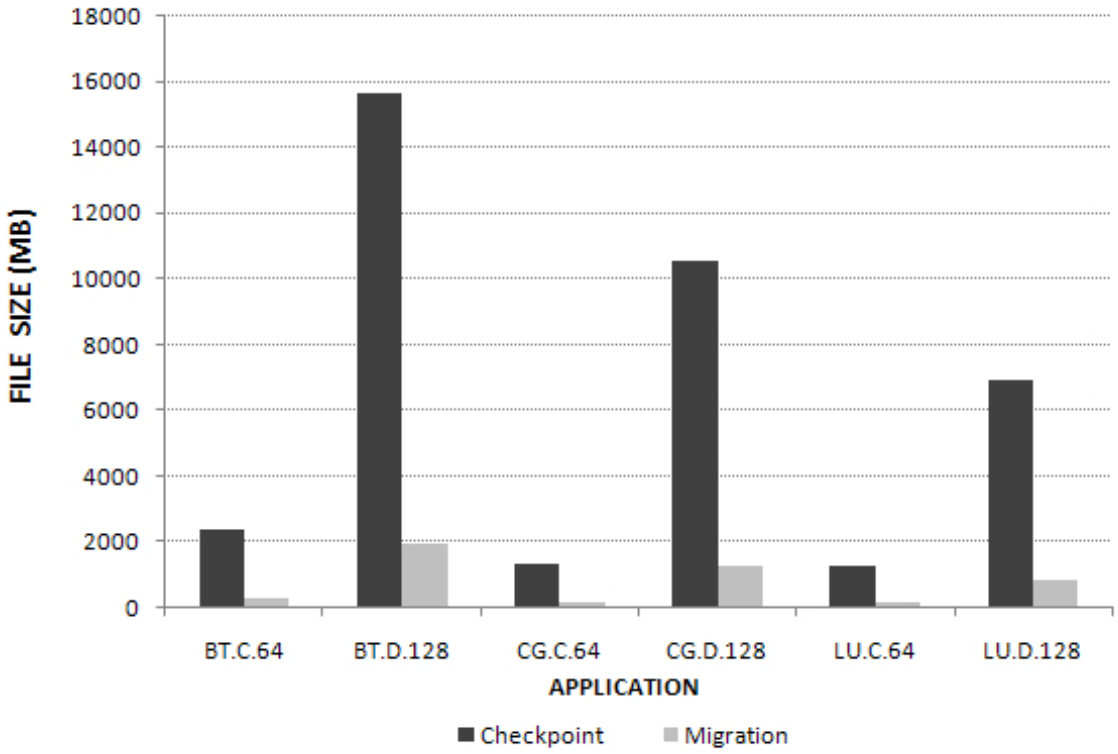


Figure 3.12: Impact of Process Migration on the File System

could be migrated away from the failing node without actually writing the images to disk.

3.5.5 Process Migration Performance

In this section, we evaluate the cost of migrating MPI Processes in the event of an impending node failure. Again, we use the LU, CG and BT applications from the NAS Parallel Benchmark Suite[45] for the evaluation. We first measure the execution time of the applications without any migration. We then measure the execution time with one migration, where we migrate all the processes of the MPI job running on one node to another spare node.

From Figure 3.13 it can be seen that impact of this migration is about 10% to 14%. About half the overhead is due to the time involved in copying the process images from the Migration Source Node to the Migration Target Node. Using a better copying scheme, or using a fast parallel file system can help alleviate this overhead.

3.6 Summary

In this chapter, we have presented a design for a Coordinated Fault Tolerance Framework for MPI which can be used to handle faults in an MPI environment. We have presented protocols for performing Checkpoint / Restart and Process Migration, as well as a framework for other FTB components, like FTB-IPMI, to cooperatively work with the Job Launcher. To the best of my knowledge, this is the first work to use the Fault Tolerant Backplane for coordinated fault handling.

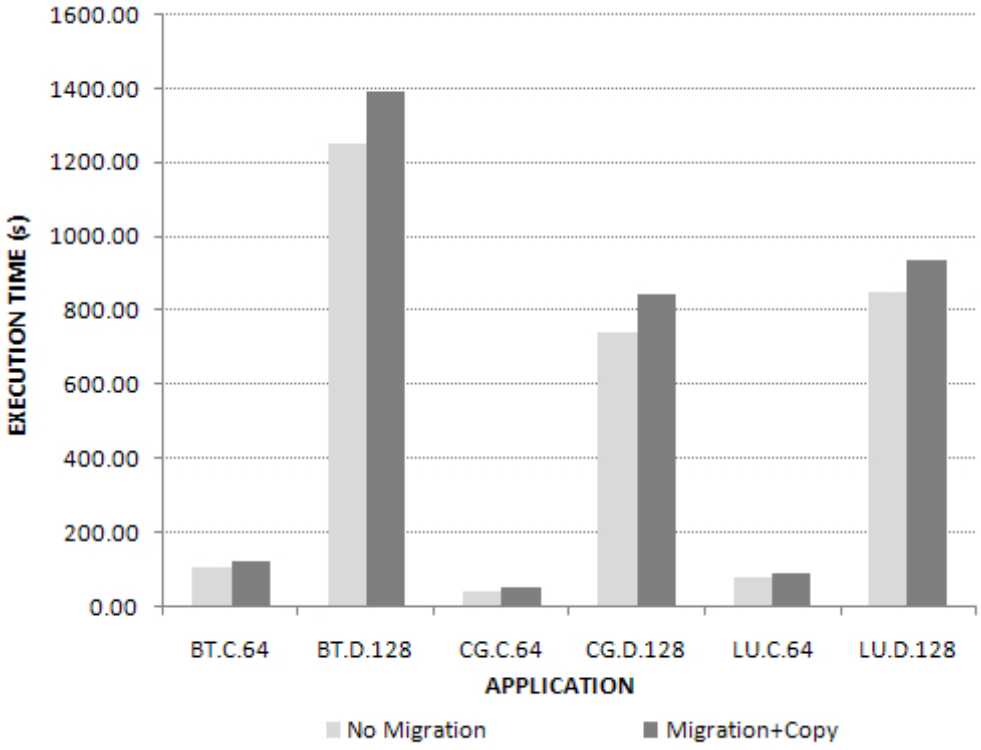


Figure 3.13: Overhead of Process Migration

CHAPTER 4

CONTRIBUTIONS AND FUTURE WORK

In this thesis, we have designed a Checkpoint / Restart framework for MPI that is interconnect-agnostic. This work greatly improves the reliability of the MPI middleware library. Our framework makes it easy for new communication channels to be integrated into the existing design. Our design also allows the flexibility of process redistribution after restart. Evaluation shows that our design provides very good performance. These designs have been available in the MVAPICH2 [11] software stack since MVAPICH2-1.2. This work can be extended by studying the performance and scalability on larger scale clusters on parallel file systems.

We have also designed a Coordinated Fault Tolerant framework for MPI that uses the Fault Tolerance Backplane. We have presented protocols to Checkpoint an MPI job, as well as Migrate processes on a node to another with very little overhead. We have also demonstrated that our design is highly extensible, where new components can be added without any modification to the existing components. In the future, we propose to incorporate an Event Driven infrastructure for the FTB-IPMI to minimize its CPU Utilization. FTB-IB has been available as an open source software since

August 2008. FTB-IPMI will be available as an open source component, as part of the MVAPICH2 release in future.

APPENDIX A

FTB-IB - LIST OF EVENTS

- **FTB_IB_ADAPTER_AVAILABLE** (INFO)

  Thrown when one or more InfiniBand HCAs are found on the system.

  Payload - "NUM_ADAPTERS:<N>"

  N - Number of HCAs found on the system

- **FTB_IB_ADAPTER_UNAVAILABLE** (WARNING)

  Thrown when no InfiniBand HCAs are found on the system.

  Payload - ""

- **FTB_IB_ADAPTER_INFO** (INFO)

  Throws information about each HCAs found on the system.

  Payload - "ADAPTER:<i>::NAME:<name>::PORTS:<P>"

  i - Index of the Adapter, ranging from 0 to (N-1)

  name - Name of the HCA

  P - Number of Ports on the Adapter

- **FTB_IB_PORT_INFO** (INFO)

  Throws information about the State of each Port on the HCA.

  Payload - "ADAPTER:<i>::PORT:<j>::STATE:<S>"

  i - Index of the Adapter, ranging from 0 to (N-1)

  j - Port of the Adapter, ranging from 1 to P

  S - State of the Port [DOWN — INIT — ARMED — ACTIVE]

- **FTB_IB_PORT_ERR** (ERROR)

  Thrown when a Port becomes unavailable on the HCA.

  Payload - "ADAPTER:<i>::PORT:<j>"

  i - Index of the Adapter, ranging from 0 to (N-1)

  j - Port of the Adapter, ranging from 1 to P

- **FTB_IB_PORT_ACTIVE** (INFO)

  Thrown when a Port becomes available on the HCA.

  Payload - "ADAPTER:<i>::PORT:<j>"

  i - Index of the Adapter, ranging from 0 to (N-1)

  j - Port of the Adapter, ranging from 1 to P

- **FTB_IB_EVENT_DEVICE_FATAL** (INFO)

  Thrown when the HCA's state transitions to FATAL.

  Payload - "ADAPTER:<i>"

i - Index of the Adapter, ranging from 0 to (N-1)

- **FTB_IB_EVENT_LID_CHANGE** (INFO)

  Thrown when the LID is changed on a Port.

  Payload - "ADAPTER:<i>::PORT:<j>"

  i - Index of the Adapter, ranging from 0 to (N-1)

  j - Port of the Adapter, ranging from 1 to P

- **FTB_IB_EVENT_PKEY_CHANGE** (INFO)

  Thrown when the Protection Key is changed on a Port.

  Payload - "ADAPTER:<i>::PORT:<j>"

  i - Index of the Adapter, ranging from 0 to (N-1)

  j - Port of the Adapter, ranging from 1 to P

- **FTB_IB_EVENT_SM_CHANGE** (INFO)

  Thrown when the Subnet Manager is changed on a Port.

  Payload - "ADAPTER:<i>::PORT:<j>"

  i - Index of the Adapter, ranging from 0 to (N-1)

  j - Port of the Adapter, ranging from 1 to P

APPENDIX B

FTB-IPMI - LIST OF EVENTS

- **FTB_IPMI_FANFAIL** (ERROR)

  Thrown when the Cooling Fan on the Processor Socket Fails.

  Payload - "<node>"

  node - Hostname of the Node where the failure is detected

- **FTB_IPMI_OVERTEMP** (ERROR)

  Thrown when the Thermal Margin on a Processor deteriorates.

  Payload - "<node>"

  node - Hostname of the Node where the failure is detected

- **FTB_IPMI_NORMAL** (INFO)

  Thrown when the node regains its health.

  Payload - "<node>"

  node - Hostname of the Node where the failure is detected

BIBLIOGRAPHY

[1] CIFTS Web Page. http://www.mcs.anl.gov/research/cifts/.

[2] CP2K Home Page. http://cp2k.berlios.de/.

[3] CPMD Consortium. http://www.cpmd.org/.

[4] Fault Tolerance Backplane - InfiniBand (FTB-IB). http://nowlab.cse.ohio-state.edu/projects/ftb-ib/index.html.

[5] InfiniBand Architecture Specification. http://www.infinibandta.com.

[6] Intel Cluster Toolkit 3.1. http://www.intel.com/cd/software/products/asmo-na/eng/cluster/clustertoolkit/219848.htm.

[7] Intelligent Platform Management Interface (IPMI). http://www.intel.com/design/servers/ipmi/.

[8] ipmiutil - IPMI Management Utilities. http://ipmiutil.sourceforge.net/.

[9] LS-DYNA Finite Element Software. http://www.lstc.com/lsdyna.htm.

[10] MPICH2: High-performance and Widely Portable MPI. http://www.mcs.anl.gov/research/projects/mpich2/.

[11] MVAPICH2: MPI over InfiniBand and 10GigE/iWARP. http://mvapich.cse.ohio-state.edu/.

[12] NAMD Home Page. http://www.ks.uiuc.edu/Research/namd/.

[13] The Basic Linear Algebra Communication Subprograms (BLACS) Package. http://www.netlib.org/blacs/.

[14] The Linear Algebra Package (LAPACK). http://www.netlib.org/lapack/.

[15] TOP500 Supercomputing Sites. http://www.top500.org.

[16] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[17] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cecile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vincent Neri, and Anton Selikhov. MPICH-V: toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[18] Aurélien Bouteiller, Franck Cappello, Thomas Herault, Géraud Krawezik, Pierre Lemarinier, and Frédéric Magniette. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25, Washington, DC, USA, 2003. IEEE Computer Society.

[19] Aurelien Bouteiller, Boris Collin, Thomas Herault, Pierre Lemarinier, and Franck Cappello. Impact of event logger on causal message logging protocols for fault tolerant MPI. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers*, page 97, Washington, DC, USA, 2005. IEEE Computer Society.

[20] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. *SIGPLAN Not.*, 38(10):84–94, 2003.

[21] Greg Bronevetsky, Keshav Pingali, and Paul Stodghill. Experimental evaluation of application-level checkpointing for OpenMP programs. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 2–13, New York, NY, USA, 2006. ACM.

[22] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[23] Camille Coti, Thomas Herault, Pierre Lemarinier, Laurence Pilard, Ala Rezmerita, Eric Rodriguez, and Franck Cappello. Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant MPI. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 127, New York, NY, USA, 2006. ACM.

[24] Duell, J., Hargrove, P., and Roman, E. Requirements for Linux Checkpoint/Restart. Technical Report LBNL-49659, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, 2002.

[25] Duell, J., Hargrove, P., and Roman, E. The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Technical Report LBNL-54941, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, 2002.

[26] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[27] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. J. Dongarra. Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems. In *Proceeding of International Supercomputer Conference (ICS)*, Heidelberg, Germany, 2003.

[28] Future Technologies Group (FTG). http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml.

[29] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[30] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K. Panda. Application-transparent checkpoint/restart for MPI programs over infiniband. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 471–478, Washington, DC, USA, 2006. IEEE Computer Society.

[31] R. Gupta, P. Beckman, B.H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra. CIFTS: A Coordinated Infrastructure for Fault-Tolerant Systems. In *Proceedings of Int'l Conference on Parallel Processing (ICPP '09)*, 2009.

[32] I.R. Philp. Software failures and the road to a petaflop machine. In *First Workshop on High Performance Computing Reliability Issues (HPCRI)*, February 2005.

[33] Hyungsoo Jung, Dongin Shin, Hyuck Han, Jai W. Kim, Heon Y. Yeom, and Jongsuk Lee. Design and implementation of multiple fault-tolerant MPI over myrinet. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 32, Washington, DC, USA, 2005. IEEE Computer Society.

[34] K. Gopalakrishnan and L. Chai and W. Huang and A. Mamidala and D. K. Panda. Efficient Checkpoint/Restart for Multi-Channel MPI over Multi-core Clusters. Technical Report OSU-CISRC-5/09-TR22, Department of Computer Science and Engineering, The Ohio State University, Columbus, OH, May 2009.

[35] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant MPI. In *CLUSTER '04: Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 115–124, Washington, DC, USA, 2004. IEEE Computer Society.

[36] Amith R. Mamidala, Rahul Kumar, Debraj De, and D. K. Panda. MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *CCGRID '08: Proceedings of the 2008 Eighth IEEE International Symposium on Cluster Computing and the Grid*, pages 130–137, Washington, DC, USA, 2008. IEEE Computer Society.

[37] MPI-Forum. http://www.mpi-forum.org/.

[38] Jun Peng, Jinchi Lu, Kincho H. Law, and Ahmed Elgamal. ParCYCLIC: Finite element modeling of earthquake liquefaction response on parallel computers. In *International Journal for Numerical and Analytical Methods in Geomechanics*, pages 1207–1232, 2004.

[39] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[40] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479–493, Winter 2005.

[41] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Jeffrey M. Squyres and Andrew Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[43] J. Sridhar, M. Koop, J. Perkins, and D. K. Panda. ScELA: Scalable and Extensible Launching Architecture for Clusters. In *International Conference in High Performance Computing (HiPC08)*, December 2008.

[44] Chao Wang, Frank Mueller, Christian Engelmann, and Stephen Scott. A Job Pause Service under LAM/MPI+BLCR for Transparent Fault Tolerance. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '07)*, 2007.

[45] Frederick C. Wong, Richard P. Martin, Remzi H. Arpaci-Dusseau, and David E. Culler. Architectural requirements and scalability of the NAS parallel benchmarks. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 41, New York, NY, USA, 1999. ACM.