

5-10-2003

Incorporating Fault-Tolerant Features into Message-Passing Middleware

Rajanikanth Reddy Batchu

Follow this and additional works at: <https://scholarsjunction.msstate.edu/td>

Recommended Citation

Batchu, Rajanikanth Reddy, "Incorporating Fault-Tolerant Features into Message-Passing Middleware" (2003). *Theses and Dissertations*. 2679.

<https://scholarsjunction.msstate.edu/td/2679>

This Graduate Thesis - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

INCORPORATING FAULT-TOLERANT FEATURES INTO
MESSAGE-PASSING MIDDLEWARE

By

Rajanikanth Reddy Batchu

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

May 2003

Copyright by
Rajanikanth Reddy Batchu
2003

INCORPORATING FAULT-TOLERANT FEATURES INTO
MESSAGE-PASSING MIDDLEWARE

By

Rajanikanth Reddy Batchu

Approved:

Anthony Skjellum
Associate Professor of Computer Science
and Engineering
(Major Professor)

Donna S. Reese
Associate Professor of Computer Science
and Engineering
(Committee Member)

Rainey Little
Associate Professor of Computer Science
and Engineering
(Committee Member)

Susan M. Bridges
Professor of Computer Science
and Engineering
(Graduate Coordinator)

A. Wayne Bennett
Dean of the College of Engineering

Name: Rajanikanth Reddy Batchu

Date of Degree: May 10, 2003

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Anthony Skjellum

Title of Study: INCORPORATING FAULT-TOLERANT FEATURES INTO
MESSAGE-PASSING MIDDLEWARE

Pages in Study: 105

Candidate for Degree of Master of Science

The popularity of MPI-based middleware and applications has led to their wide deployment. Such systems, however, are not inherently reliable and cannot tolerate external faults. This thesis presents a novel model-based approach for exploiting application features and other characteristics to categorize and create AEMs (Application Execution Model). This work realizes MPI/FT™, a middleware derived by selective incorporation of fault-tolerant features into MPI/Pro™ for two relevant AEMs.

This thesis proves the following hypothesis: it is possible to successfully complete select MPI applications even in the presence of external faults, and such fault-tolerance can be achieved with acceptable performance overhead. This work defines parameters to measure the impact of this middleware on performance through fault-free and fault-

injected overheads. The hypothesis is validated through experimentation and measurement of sample MPI applications for two AEMs.

DEDICATION

To my loving family.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my major advisor Dr. Anthony Skjellum. This work would not have been possible without his guidance and suggestions. I would also like to thank my committee members Dr. Reese and Dr. Little for their suggestions. I also thank Mr. Yogi Dandass for his insightful suggestions and mentoring.

I would like to thank Dr. Beddhu and Dr. Dimitrov for their valuable suggestions during my internship at MPI Software Technology Inc, where most of the ideas for this research were formed. I would also like to thank my colleagues at the HPC lab, who made it more fun on every step of the way.

TABLE OF CONTENTS

	Page
DEDICATION	ii
ACKNOWLEDGEMENTS	iii
LIST OF TABLES	vii
LIST OF FIGURES	viii
CHAPTER	
I. INTRODUCTION	1
1.1 Background	1
1.2 Motivation	2
1.3 Hypothesis	8
1.4 Basis	9
1.5 Contributions	9
1.6 Organization	10
II. LITERATURE SURVEY	11
2.1 Fault-tolerance Basics	11
2.2 Limitations of MPI standard	12
2.3 Other Research efforts	14
2.3.1 CoCheck	14
2.3.2 Egida	15
2.3.3 FT-MPI	16
2.3.4 Starfish	16
2.4 Summary	17
III. MODEL-BASED APPROACH	18
3.1 Research Basis	18
3.2 Model-based Parameters.....	21

CHAPTER	Page
3.3 Application Execution models.....	23
3.3.1 Model-Ia	24
3.3.2 Model-IIa	26
IV. APPROACH	29
4.1 Research Methodology	29
4.2 Usage	30
4.2.1 User level changes	30
4.2.2.1 Development and steps	31
4.2.2.2 FT API and code modifications	34
4.3 Design and Implementation	44
4.3.1 Detection	45
4.3.2 Notification and Recovery	50
4.3.2.1 Model-Ia Recovery	50
4.3.2.2 Model-IIa Recovery	52
V. EXPERIMENTS, RESULTS, AND ANALYSIS	55
5.1 Message-passing Overheads	56
5.1.1 Latency Overhead	57
5.1.1.1 Experimental Setup	57
5.1.1.2 Results and Analysis	58
5.1.2 Bandwidth Overhead	61
5.1.2.1 Experimental Setup	61
5.1.2.2 Results and Analysis	63
5.2 Model-Ia Results	64
5.2.1 Runtime Overhead	64
5.2.1.1 Experimental Setup	65
5.2.1.2 Results and Analysis	66
5.2.2 Program changes	67
5.2.2.1 Program Change Ratio	67
5.2.2.2 Results and Analysis	67
5.2.3 Recovery Time	68
5.2.3.1 Experimental Setup	68
5.2.3.2 Results and Analysis	69
5.3 Model-IIa Results	72
5.3.1 Runtime Overhead	73
5.3.1.1 Experimental Setup	73
5.3.1.2 Results and Analysis	74
5.3.2 Program Changes	76
5.3.2.1 Program Change Ratio	76
5.3.2.2 Results and Analysis	76

CHAPTER	Page
5.3.3 Recovery Time	77
5.3.3.1 Experimental Setup	77
5.3.3.2 Results and Analysis	78
5.4 Summary	79
VI. OTHER OBSERVATIONS	81
6.1 Components of Fault-free Overhead	81
6.2 Placement of Services	85
VII. CONCLUSIONS	88
7.1 Summary	88
7.2 Future work	90
REFERENCES	92
APPENDIX	
A. OTHER MODEL-IA RESULTS	95
B. SCALABILITY TESTS	101

LIST OF TABLES

TABLE	Page
1.1 Hypothesized Parameter values for Model-Ia and Model-II	9
3.1 Application Execution Models	23
5.1 Fault-free Runtime Overhead in Model-Ia, Pmandel Application	67
5.2 Program Change Ratio in Model-Ia, Pmandel Application	67
5.3 Recovery Time in Model-Ia, Pmandel Application	70
5.4 Fault-free Runtime Overhead without Checkpointing in Model-IIa, Game of Life Application	75
5.5 Program Change Ratio for Model-IIa, Game of Life Application	77
5.6 Middleware Level Recovery Time for Model-IIa, Game of Life Application	78
5.7 Application Level Recovery Time for Model-IIa, Game of Life Application	78
B.1 Speedup of Game of Life Application with MPI/Pro and MPI/FT	103

LIST OF FIGURES

FIGURE	Page
1.1 Probability of Completion in Harsh Environments	3
1.2 Typical Terminate-and-Restart Strategy	4
1.3 Mean Runtimes under Terminate-Restart and Fall through Fault-tolerant Strategy for Successful Completion	6
3.1 Middleware Design Process Inputs	21
3.2 Model-Ia: Maser-Worker Application Model-Ia	25
3.3 Model-IIa: SPMD all-interacting model	27
4.1 Utilization of Spare Ranks in MPI/FT	34
4.2 Pseudo-code for Unmodified Model-Ia Application	36
4.3 Pseudo-code for Modified Model-Ia Application	37
4.4 Pseudo-code for Unmodified Model-IIa Application	41
4.5 Pseudo-code for Modified Model-IIa Application	43
4.6 Primary Steps in Achieving Fault-tolerance	45
4.7 Coordinator and SCT: External heartbeat mechanism	47
4.8 SCT and Progress Threads: Internal heartbeat mechanism	49
4.9 Notification and Recovery in Model-Ia	51
4.10 Notification and Recovery in Model-IIa	54
5.1 Pseudo-code for measuring Latency	58

FIGURE	Page
5.2 Latency Overheads with only External Heartbeats	59
5.3 Latency Overheads with Internal and External Heartbeats	59
5.4 MPI/Pro Latency and Latency Overheads for smaller message sizes	61
5.5 Pseudo-code for measuring Bandwidth	62
5.6 Bandwidth Overheads with only External Heartbeats	63
5.7 Bandwidth Overheads with Internal and External Heartbeats	64
5.8 Pseudo-code for Model-Ia Application	66
5.9 Runtime of Pmandel Application with faults at 10 % of Progress	71
5.10 Runtime of Pmandel Application with faults at 90 % of Progress	71
5.11 Pseudo-code for Model-IIa Application	74
5.12 Fault-free Overhead with Checkpointing in Model-IIa, Game of Life Application	76
6.1 Fault-free Overhead and CC-Ratio for MPI/FT Model-Ia at 40 bytes	83
6.2 Fault-free Overhead and CC-Ratio for Debug-enabled MPI/FT Model-Ia at 40 bytes	84
7.1 Model-Ic with Parallel NMR	91
A.1 Pmandel Application Runtime with Fault at 20 % Progress	97
A.2 Pmandel Application Runtime with Fault at 30 % Progress	97
A.3 Pmandel Application Runtime with Fault at 40 % Progress	98
A.4 Pmandel Application Runtime with Fault at 50 % Progress	98
A.5 Pmandel Application Runtime with Fault at 60 % Progress	99
A.6: Pmandel Application Runtime with Fault at 70 % Progress	99
A.7 Pmandel Application Runtime with Fault at 80 % Progress	100

FIGURE	Page
B.1 Speedup of Game of Life, 16x16	104
B.2 Speedup of Game of Life, 250x250	104
B.3 Speedup of Game of Life, 1Kx1K	105

Chapter I

INTRODUCTION

1.1 Background

Clusters of COTS (Commodity Off The Shelf) components are rapidly replacing traditional supercomputers. Clusters are networks of workstations interconnected by high-speed networks. Recent advances in individual processors and interconnect technologies have made clusters more reliable, scalable, and affordable [9]. They have been the solution of choice for problem solving in several domains that require large amounts of computation power.

The efficacy of these clusters for parallel computing is determined by two factors, namely the middleware and the parallel programming environment. Middleware typically glues various components in the systems and provides an abstract view of the system to its users. Parallel programming environments provide a programming interface for developing parallel applications. MPI (Message Passing Interface) [24] is a standard for message-oriented middleware that also provides a parallel programming interface. MPI's main goals are high performance and portability, and it is currently the *de facto* standard for message-passing libraries. Numerous implementations of MPI have been realized both in industry and academia. MPI has been successfully used in many

domains such as scientific computing and visualization [3, 7, 19, 27].

Clusters inherently increase the availability at hardware level with redundant and hot-swappable components, but these features are not automatically transferred to higher layers. With the increasing popularity of clusters and MPI applications, the issue of reliability at the middleware and application layers is gaining prominence. The MPI standard is limited and does not consist of comprehensive reliability measures. While some other research efforts have realized fault-tolerant MPI, they employ common mechanisms that may lead to high overheads and contradict the high-performance goals of MPI. This thesis realizes MPI/FT™ [4], a low-overhead, fault-tolerant message-passing middleware. MPI/FT has been realized by incorporating select fault-tolerant features in MPI/Pro™ [26], an existing high-performance realization of MPI 1.1 standard.

1.2 Motivation

Clusters and MPI-based systems have proliferated in both academic and industrial environments. These systems have been widely deployed in embedded, e-commerce/web, and production environments and are used for both critical and non-critical operations. Demand for supercomputing power in space-based missions has also necessitated the use of clusters [19]. These space-based environments typically induce external faults at a non-trivial rate. Faults and failures are also unavoidable in many ground systems. In particular the probability of a node or OS (Operating System) failure increases with the number of components. Figure 1.1 presents an intuitive diagram representing increasing difficulty for longer MPI applications to finish in the presence of faults. Failures in many of these systems are associated with high costs as they typically manifest into loss of

critical data and time. It is essential that these systems be capable of tolerating both external and internal faults and provide services with minimum disruption.

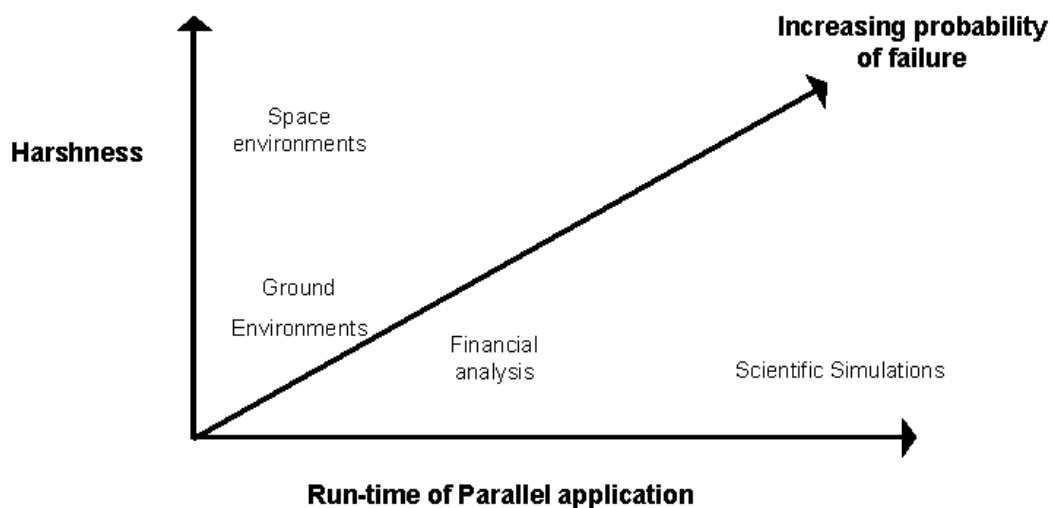


Figure 1.1: Probability of Completion in Harsh Environments

MPI's [24] main goals of high-performance and portability have led to the exclusion of comprehensive reliability measures. This lack of reliability measures is evident in the assumptions of a reliable communication layer, limited fault detection, and limited recovery procedures in the standard. MPI has also been designed to work in relatively safe environments and is typically not used in harsh environments such as space-based missions. Chapter 2 discusses these limitations in detail. MPI implementations have typically used a static process model to realize MPI, requiring successful completion of all constituent processes for completion of the application.

Current MPI implementations handle errors in processes by terminating the application, and typically users restart the application. This simplistic terminate-and-restart view is discussed in [4] and is also presented in Figure 1.2.

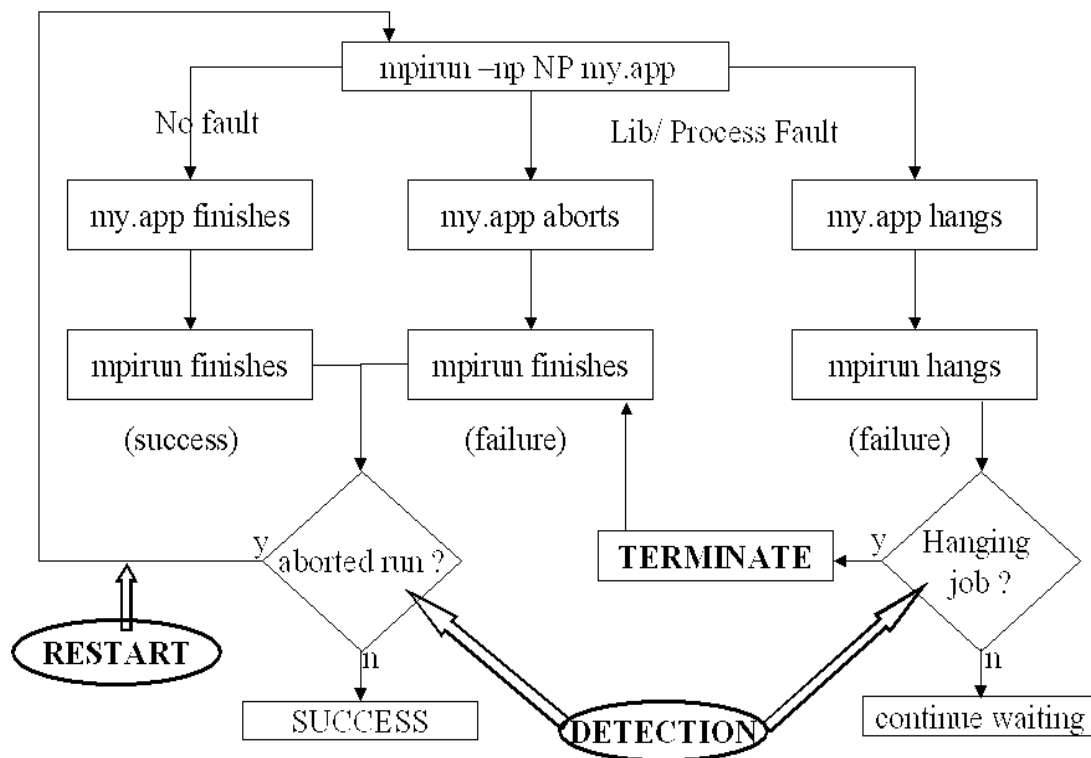


Figure 1.2: Typical Terminate-and-Restart Strategy

MPI applications are typically launched using a program launcher such as *mpirun*. In the absence of any faults both the application and *mpirun* return successfully. In the presence of a fault the application may terminate to return errors through *mpirun*. In such a case the application can be restarted through manual intervention, or the restart capability can be included in *mpirun*. However, based on the middleware implementation and the application state at the instance of occurrence of the fault, the application may

hang indefinitely instead of returning errors to mpirun. In such a case external mechanisms to detect hanging or unresponsive jobs are required.

A terminate-and-restart strategy is an unviable option even if processes are guaranteed to terminate in the presence of faults. Consider an application that takes $T_{App_Original}$ time units to successfully complete in the absence of faults. The faults in the system are expected to follow a Poisson distribution with a fault rate of λ . The mean time for the application to finish utilizing the terminate-and-restart strategy is given by equation 1.1. This time $T_{Restart_Mean}$ represents the sum of mean time the application has to wait for a fault-free zone and the original runtime of the application. The mean time before the application has to wait for a fault-free zone of duration greater than $T_{App_Original}$ can be derived using interval estimation [15, 20]. The runtime is expected to increase exponentially with the fault-rate or runtime of the application and is given by

$$T_{Restart_Mean} = \frac{e^{\lambda T_{App_Original}} - 1}{\lambda} . \quad (1.1)$$

Instead of a terminate-and-restart strategy, a “fall through” fault-tolerant strategy allows applications to detect and recover from faults. Applications need not be reinitialized and can be recovered using a combination of middleware and application-assisted mechanisms. Additional mechanisms for such fault detection and recovery are expected to increase the overall runtime. For the same application the mean runtime T_{FT_Mean} for a successful completion under a fault-tolerant strategy is presented in equation 1.2. The fault-tolerant system is assumed to impose a fault-free overhead of

$O_{Fault-Free}$ % and a mean recovery time $T_{App_Recovery}$ for each fault encountered. The mean runtime is given by

$$T_{FT_Mean} = T_{App_Original} (1 + O_{Fault-Free}) (1 + \lambda T_{App_Recovery}). \quad (1.2)$$

Figure 1.3 presents the mean runtime under both the terminate-and-restart strategy and the fall through fault-tolerant strategy for an application whose runtime is one hour. Mean runtimes are computed for the fault-tolerant case with assumed overheads of 100% and 200% and a recovery time of 5 minutes for occurrence of each fault.

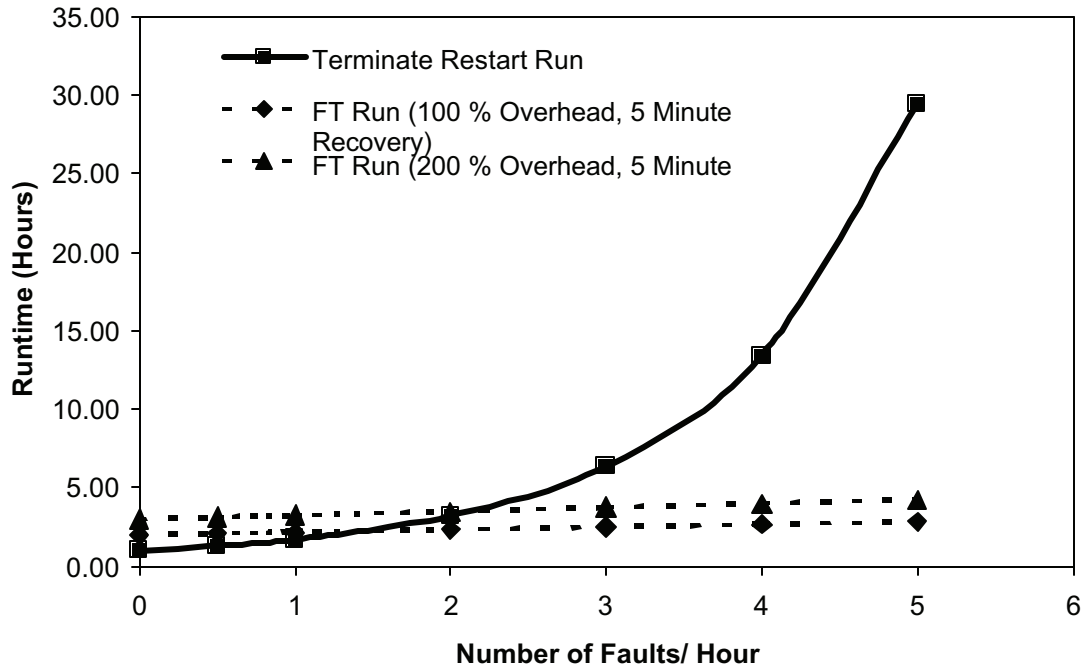


Figure 1.3: Mean Runtimes under Terminate-Restart and Fall through Fault-tolerant Strategy for Successful Completion

It is evident from the graphs that the mean runtime under the terminate-and-restart strategy is much higher than for the fault-tolerant strategy at non-trivial rates of faults.

However, at smaller fault rates the fault-tolerant strategy incurs larger runtimes than a simple terminate-and-restart strategy. Users may choose to adopt either strategy based on several factors, some of which are the fault rate, the impact of mean runtime, the cost of implementing fault-tolerance, or the costs associated with a failure. Most high-performance applications in critical environments, which are associated with high costs of failure, would be expected to adopt a fault-tolerant strategy rather than a terminate-and-restart view. It must be noted that exaggerated values for fault-free overheads and recovery time have been used for the graphs in Figure 1.3. Experiments in subsequent chapters show that low overheads in the vicinity of 10% and recovery time as the order of milliseconds is possible for some practical applications.

Thus, successful completion of an MPI application in the presence of external faults and in a harsh environment is non-trivial and requires fault-tolerance and reliability in the MPI implementation. High costs of failure in critical systems, insufficiency in the MPI standard, and limitations in current implementations of MPI necessitate the need for a fault-tolerant and reliable message-passing middleware. MPI/FT attempts to satisfy this need by providing for an effective approach and implementation.

The scope of this research is limited to specific MPI applications. These applications are assumed to be well written and devoid of internal errors arising during design and development phases. They are expected to run successfully to completion in the absence of external faults. MPI/FT aims at enabling such applications to complete successfully even in the presence of external faults.

1.3 Hypothesis

This thesis hypothesizes that a modified MPI application with a given set of application features developed on a modified MPI middleware will run successfully to completion, even in the presence of a set of modeled fault conditions and will incur acceptable fault-free performance overhead and acceptable application changes.

Application features refer to discerning characteristics, such as communication topology and application structure, that are abstracted in AEMs (Application Execution Model). This hypothesis is proved using two AEMs, namely Model-Ia and Model-IIa. Model-Ia abstracts MPI applications that follow a simple master-worker style with a star communication topology. Slave processes in this model can die and recover without stalling the entire application's progress. Model-IIa abstracts applications belonging to a SPMD (Single Program Multiple Data) style and an all-to-all communication topology. Death of a single process in this AEM forces the entire application to recovery. Parameters that differentiate amongst AEMs and assumptions and restrictions on each of the two models are further described in Chapter 3.

Performance is evaluated in terms of message-passing overheads and the run-time overhead of an application. Application changes are measured using PCR (Program Change Ratio), which is defined as a ratio between the number of new API calls introduced and the number of lines in the original code. Acceptable performance overhead and program changes were hypothesized on a per AEM basis. The acceptable values for fault-free parameters and recovery for both Model-Ia (master-worker) and Model-IIa are presented in Table 1.1.

Table 1.1: Hypothesized Parameter values for Model-Ia and Model-IIa

Parameter		Model-Ia	Model-IIa
Fault-Free Overheads	Message-Passing	5%	5%
	Run-time	15%	30%
	PCR	10%	10%
Recovery Time (milliseconds)	Middleware Recovery	25	50
	Application Recovery	500	500

1.4 Basis

Several research efforts exist to make MPI more reliable. They are briefly described in Chapter 2, the literature survey. Most of these efforts treat the applications as a "black box" and provide the same measures for reliability. Some such measures are user-transparent checkpointing, and rollback and recovery measures. This thesis and research are based on the fundamental premise that MPI-based parallel applications can provide features and characteristics that are amenable to achieving fault-tolerance and reliability. The basis of this research is that such exploitation of application features will yield fault-tolerance at lower overheads. These discerning characteristics and relevant fault-tolerant features are coupled into AEMs. Chapter 3 presents these features and explains relevant models.

1.5 Contributions

The contributions of this thesis are as follows:

- 1) This thesis introduces a new model-based approach for exploiting application features for achieving low overhead fault-tolerance.

- 2) This work has identified several practical AEMs based on the model-based approach.
- 3) This work has realized prototype implementations of MPI/FT for two prominent AEMs, Model-Ia (simple master/slave) and Model-IIa (SPMD).
- 4) This thesis has identified and measured fault-free and fault-injected parameters to understand the impact of achieving fault-tolerance in message-passing middleware on performance. It also shows that that fault-tolerance can be achieved with low fault-free overheads.

1.6 Organization

The remainder of the document is organized as follows. Chapter 2 presents the limitations in MPI standard and inadequacies in various MPI implementations based on a literature review of various research efforts to make MPI reliable. Chapter 3 presents the model-based approach and presents two AEMs: Model-Ia and Model-IIa. Chapter 4 presents the research approach for this thesis. Chapter 5 presents the parameters to validate this thesis and experiments to obtain these values. Chapter 6 presents information about observations and experiments to provide more insight. Chapter 7 concludes the document and suggests future work.

Chapter II

LITERATURE SURVEY

This chapter presents a literature review of topics essential to this research. A summary of fault-tolerance basics is presented in section 2.1. Section 2.2 discusses the shortcomings and limitations in MPI-1.1 standard [24]. Section 2.3 briefly introduces other research efforts and implementations to make more MPI more reliable and discusses their drawbacks and limitations.

2.1 Fault-tolerance Basics

Dependability is an essential quality of systems. It can be defined as reliance on a system to deliver services [17]. A system can be considered dependable when it is available, reliable, and safe. Availability is defined as the probability that a system can offer its services at a given instance of time. Reliability is defined as the percentage of time a system conforms to its specifications and provides services. Safety is defined as a system's ability to operate safely and avoid catastrophic results.

Errors, faults, and failures refer to the same fundamental problem of deviations from specifications at different levels of abstractions. These terms are used interchangeably in the rest of this document. Faults can be classified by their origin, effects, and duration.

Various ways of achieving reliability [17], and hence dependability, in systems include:

- Fault avoidance: Eliminates introduction of faults at design stages. Typically achieved by verification and validation techniques.
- Fault removal: Removes faults during the testing stage. Includes techniques like white box and black box testing.
- Fault-tolerance: Accepts the fact that faults cannot be avoided in several systems. Relies on mechanisms to detect and recover from faults during runtime.
- Fault Evasion: Reduces introduction of faults during runtime of the system. Achieved by altering system parameters such as input load and resource usage.

Fault-tolerance is the prevalent way to achieve reliability, as faults cannot be completely removed during design and testing phases, and external faults cannot be prevented. Fault-tolerance achieves reliability through the use of redundant components. Redundancy of components can be spatial, temporal, or informational. Redundancy can be either in active or passive mode. Active redundancy is marked by multiple replicas working concurrently to mask faults. Passive redundancy is marked by a single active component that is backed up by several passive replicas. These passive replicas take over after loss of the active component. Fault-tolerance typically includes several steps such as detection, diagnosis, isolation, repair, and recovery.

2.2 Limitations of MPI standard

MPI forum released the first MPI standard, MPI-1.1 [24], in 1995. The main goals of the standard in this release were high performance and portability. Achieving reliability typically includes utilization of additional resources and methodology. This additional utilization conflicts with the main goal of high performance and supports the

MPI forum's decision for limited reliability measures. Emphasis on high performance thus led to a static process model with limited error handling capabilities and reliability. The success of an MPI application is guaranteed only when all the constituent processes finish successfully. Death or crash of one or more processes leads to a default application termination. Users are typically expected to restart the application manually. This simplistic terminate-and-restart view is not capable of providing required reliability in several environments, and its drawbacks are discussed in Chapter 1.

Current implementations of MPI suffer inadequacies in various respects to providing reliability.

- **Fault Model:** MPI standard [24] and current implementations of MPI assume a limited fault model. MPI assumes a reliable communication layer and does not provide mechanisms for dealing with an unreliable communication layer. The standard does not provide methods to deal with node failures and lost messages. MPI also limits faults in the system to incorrect parameters in function calls and resource errors. This coverage of faults is incomplete and insufficient for systems placed in harsh environments.
- **Fault Detection:** Fault detection is not defined by MPI. MPI provides for limited fault notification in the form of return codes in functions. Critical faults, such as process crashes, may preempt functions from returning these return codes to the caller level.
- **Fault Recovery:** MPI provides users with functions to register error-handling callback functions. These callback functions are invoked by the MPI implementation in the event of an error in MPI functions. Callback functions are

registered on a per communicator basis and do not allow for per function based error handlers. Callback functions provide limited capability and flexibility and cannot be invoked in case of process crashes and hangs.

MPI forum released the MPI-2 [25] standard in 1998. MPI-2 consists of extensions in the areas of process creation and management, one-sided communications, extended collective operations, and parallel I/O. A significant contribution of MPI-2 is the DPM (Dynamic Process Management), which allows user programs to create and terminate additional processes on demand. DPM may be used to compensate for the loss of a process, but the lack of detection and recovery precludes reliability.

2.3 Other Research efforts

Several research efforts have been conducted to make MPI implementations more reliable. This section introduces some of these efforts and analyzes their drawbacks and shortcomings in providing for a reliable MPI Middleware. Solutions in providing reliable MPI have ranged from transparent checkpointing and emphasis on health of the communicator to utilizing MPI-2's DPM.

2.3.1 CoCheck

CoCheck [30] is one of the earliest efforts to make MPI more reliable. CoCheck extends the single process checkpoint mechanism in Condor [22] to a distributed message-passing application. Unlike most checkpointing middlewares, CoCheck is visible to the user and is available at a layer above the message-passing middleware. Problems in checkpointing, such as global inconsistent states and domino effects, are eliminated by the usage of a flush protocol. This user-aware flush protocol sends “ready

messages” to all processes. Receipt of this “ready message” causes purging of message buffers and clearing of communication channels. CoCheck was primarily targeted for process migration, load balancing, and stalling long-running applications for resumption at a later time.

CoCheck incurs a large overhead by checkpointing entire process state. Recovery of a dead process is achieved by a recovery function run at user level, but this is insufficient. The status of inconsistent internal data structures in message-passing middleware is not addressed. Checkpointing is also not a viable option for certain MPI applications, such as those following the Master-Worker model. Applications of this type have a simple model where a master process distributes jobs among worker processes, and workers return results. In such a model, checkpointing the state of workers is unnecessary as it can be reconstructed from the saved jobs from the master process. Thus, CoCheck provides for coarse reliability measures for MPI.

2.3.2 Egida

Egida [29] is an object-oriented toolkit for transparent rollback and recovery. Egida is extensible and allows users to define their own rollback recovery protocols. Implementations for the described protocol are synthesized by gluing pre-existing objects. Egida bases itself on log-based rollback recovery protocols and mainly emphasizes low overhead during recovery and rollback. This checkpointing and rollback of messages is transparent to the user. Egida has been ported to MPICH [14], an academic implementation of MPI. The Egida layer has been placed between the higher MPICH layer and the p4 communication layer. Modifications have been made to include a watchdog timer and to allow socket reestablishment in case of process failures.

Applications need to relink with Egida to achieve transparent fault tolerance. Egida shares some of its drawbacks with CoCheck. Egida checkpoints both processes and messages, which may lead to large overheads in some cases.

2.3.3 FT-MPI

A communicator is an important data structure defined in the MPI standard [24]. A communicator defines a communication context, usually denoted by an identifier, and a set of processes in the context. Communicators are essential for maintaining different communication contexts. FT-MPI [13] acknowledges that the health of a communicator is essential for proper running of an MPI application. The death of processes places communicators in an inconsistent state. FT-MPI suggests expanding and shrinking communicators in lieu of process deaths and inclusions, and it emphasizes methods to have redundant slots for new processes and various ways of managing the communicator data structure. FT-MPI does not take care of detection and recovery at the user level.

2.3.4 Starfish

Starfish [1], from Technion University, Israel, is a partial MPI-2 implementation with DPM. The Starfish environment for execution of static and dynamic MPI programs is based on the Ensemble group communication system [16]. Starfish provides hooks to handle dynamic cluster changes and checkpointing. It uses an event model where processes and components register to listen on events. This event bus provides messages reflecting changes in cluster configuration and process failures. Starfish introduces a novel object bus for event dissemination.

Starfish provides fault-tolerance as a byproduct, but it does not provide for user level recovery API, and the consistency of communicator and internal data structures is not addressed.

2.4 Summary

This chapter has presented the shortcomings of MPI standard and the various attempts to make it more reliable. These inadequacies motivate the need for a fault-tolerant middleware. The essential features of such a middleware would be low overhead and adaptability. These features would be essential for a middleware to successfully cater to different applications under different fault conditions.

Chapter III

MODEL-BASED APPROACH

This chapter presents the basis for the research presented in this thesis. The model-based approach presents a way of exploiting application features to achieve low-overhead fault-tolerance. This chapter explains this approach and introduces parameters that separate various AEMs. Based on the model-based approach two AEMs or models, Model-Ia and Model-IIa are described.

3.1 Research Basis

Middleware is a class of software geared towards managing complexity [2]. Middleware typically provides API for the users and applications at higher levels by abstracting lower level details. Middleware is also known informally as the “gluing” or “plumbing” component that connects and passes data. Middlewares are typically designed and developed to support a range of applications with different characteristics.

MPI [24] is a message-oriented middleware that also provides a parallel programming interface. As described in the literature survey, several other efforts have tried to make MPI-based middlewares more reliable. Many of these middlewares provide similar fault-tolerant services to all applications. Some such services available that

achieve at fault-tolerance and reliability are user transparent checkpointing, message-logging and roll-back and recovery. This "black box" approach of providing similar services may incur large overheads in some applications.

The model-based approach is based on the premise that applications consist of features that are amenable to and aid in achieving low-overhead fault-tolerance. Different applications require different services to achieve fault-tolerance, and the model-based approach determines these services based on different classes of applications. Each class of applications is distinguished by various parameters and gives rise to AEMs. The parameters for the model-based approach and relevant AEMs are explained in subsequent sections. It is expected that an MPI middleware will be realized for each of these models and that such an approach will lead to realization of low-overhead fault-tolerance.

Apart from providing differentiated services, the model-based approach also aims at managing the middleware design complexity. The middleware design process is inherently complex, and the additional requirement of reliability exacerbates this design process. The complexity of the design process arises from both the numerous factors that need to be considered and the contrasting requirements placed by each of these factors. These relevant factors are listed below and presented in Figure 3.1.

- **Fault Model:** A fault model defines and limits the faults possible in the system, along with their possible manifestations. A system designed with a particular fault model can be guaranteed to work only under a subset of the defined fault model. A fault model greatly influences the design and implementation of effective reliability measures and would primarily determine the range and strength of fault detection mechanisms.

- **Underlying layers:** Middleware provides an abstraction over the underlying hardware and OS (Operating System). Features and reliability measures offered by these layers have a direct influence on the type of services offered by the middleware. Some hardware components are capable of providing safe execution environments to processes, and several OSs provide additional reliability features. Design of the middleware can be optimized to make use of these features. These underlying features would determine the amount of redundancy possible in the system and the physical placement of processes. For example, embedded systems with limited memory resources are more amenable to passive redundancy than active redundancy.
- **Application features:** A wide range of applications can be run on a middleware. Applications written over a message-passing middleware can be segregated by their communication topology, communication patterns and resource usage. This segregation can allow for specialization of services provided to a class of applications and thus leads to a more flexible middleware. In comparison with a monolithic middleware that relies on a “one size fits all” approach, an application class-based middleware may be more streamlined and provide better performance.
- **Application goals:** Application goals can be categorized under runtime and reliability goals. Runtime goals can be focused on either high performance or predictability. High performance systems emphasize minimizing the runtime of the application, while predictability is an essential quality for running real-time applications and requires guarantees of bounded detection and recovery time. Reliability and availability requirements determine the physical location of certain

processes in “safe” environments if possible and the amount and type of redundancy required. Applications that require high reliability and availability may choose to have additional redundancy and complex management policies.

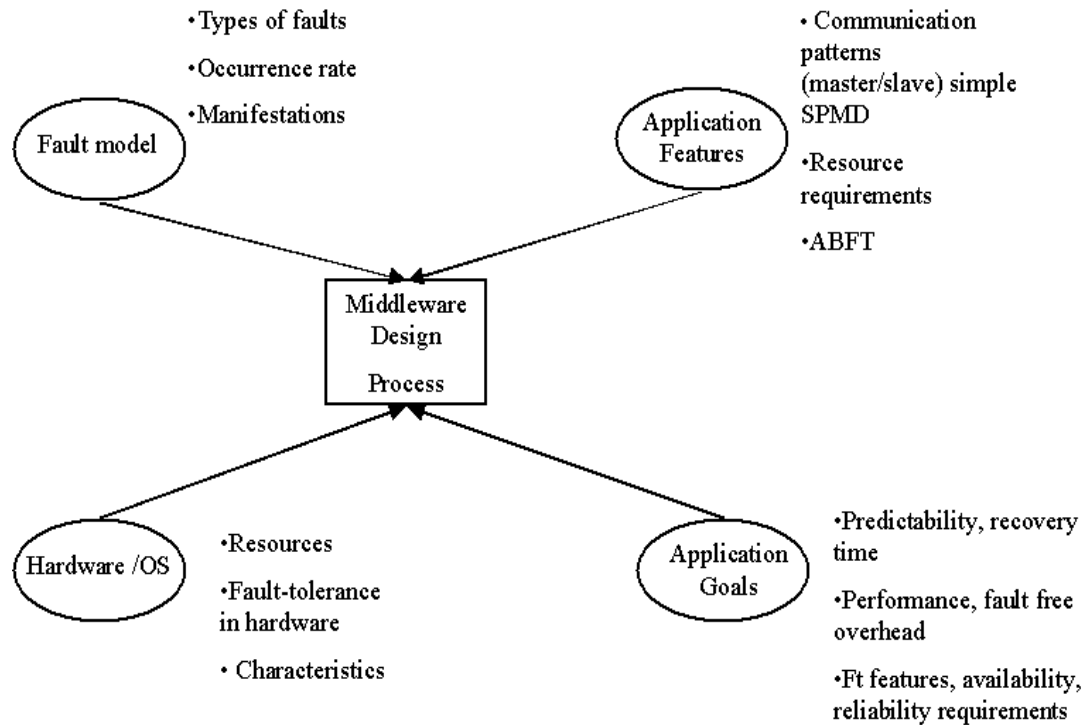


Figure 3.1 : Middleware Design Process Inputs

The model-based approach, by providing abstractions across these features and requirements, alleviates the complexity issues.

3.2 Model-based Parameters

The model-based approach solves the problems associated with the typical "black box" approach and aims at reducing overheads for fault-tolerance by providing tailored

services based on applications. This section presents the parameters that distinguish various AEMs. They are as follows:

- **Virtual Communication Topology:** MPI-based applications consist of a virtual topology, which is determined by the connectivity of various processes and the message transfers in the parallel application. Commonly used topologies in parallel applications include star, all-to-all fully connected, rectangular grid, and hypercube. Such a virtual topology may be different from instantiated MPI communicators. For example, each process may frequently communicate within a subset of neighbors or a stencil, while infrequently utilizing its default all inclusive `MPI_COMM_WORLD`. This communication topology affects availability and implementation of collective calls, application-based detection, and middleware level recovery.
- **Program Style:** Application requirements determine the style of MPI programs and commonly follow either a single program multiple data (SPMD) or master-worker style. The programs are typically written in iterative loops with reoccurring patterns of communication, computation and interspersed with global barriers. These program patterns determine appropriate placement and usage of MPI/FT API in the user code.
- **Middleware Redundancy:** Replication of both process and data are essential for achieving fault-tolerance and hence reliability. Effective management of redundancy is essential for achieving low-overhead fault tolerance. In a master-worker model, redundancy is required primarily by the master rather than the worker. Worker processes have a minimal state to remember and hence can be

restarted quickly. In embedded systems, passive redundancy is preferred over active redundancy to optimize resource usage.

3.3 Application Execution models

Table 3.1: Application Execution Models

Programming Style	Communication Topology	Middleware Redundancy	AEM Designation	Currently Implemented
Master/Worker	Star	None	Model-Ia	Yes
		Master -passive	Model-Ib	No
		Master-active	Model-Ic	No
SPMD	All-to-all	None	Model-IIa	Yes
		Rank 0- passive	Model-IIb	No
		Rank 0- active	Model-IIc	No

Table 3.1 lists several possible AEMs for the master-slave and SPMD styles. Two AEMs, Model-Ia and Model-IIa, are implemented for MPI/FT and are explained in detail. These two models are widely used and hence represent a large set of MPI parallel applications. Assumptions for each model and the features services to support fault - tolerance are described. Coordinator and SCT (Self Checking Thread) are middleware level threads that enable fault detection and recovery. These concepts are explained in detail in section 4.3.1.

3.3.1 *Model-Ia*

Master-worker is a simple process model where the master divides and distributes the jobs among a set of worker processes. Workers operate on the job and return the results to the master. In this model, master and workers share a virtual star topology with the master at the center. Figure 3.2 presents Model-Ia. The assumptions and features for this model are as follows.

Assumptions:

The model is simple and involves message-passing between master and worker and not between workers. There is no explicit synchronization among workers or globally, and collective calls are disallowed because of the star-topology. Faults are expected to affect the workers, and the master process is assumed to be in a safe process area free from external faults. A master process can tolerate the death of a worker process, but death of the master cannot be tolerated. In Model-Ia death of a master implies that the entire progress of the application is lost, and the application needs to be restarted. AEMs Model-Ib and Model-Ic are aimed at alleviating this situation and are currently under investigation .

Applications are expected to be written in an iterative manner at the master process, where the master sends and receives jobs to the workers. Each iteration typically consists of the master process receiving results from a worker and then assigning a new job to the worker. In the event of the death of a worker, all MPI communication to that worker must be terminated with errors, but MPI communication with other alive workers is still valid.

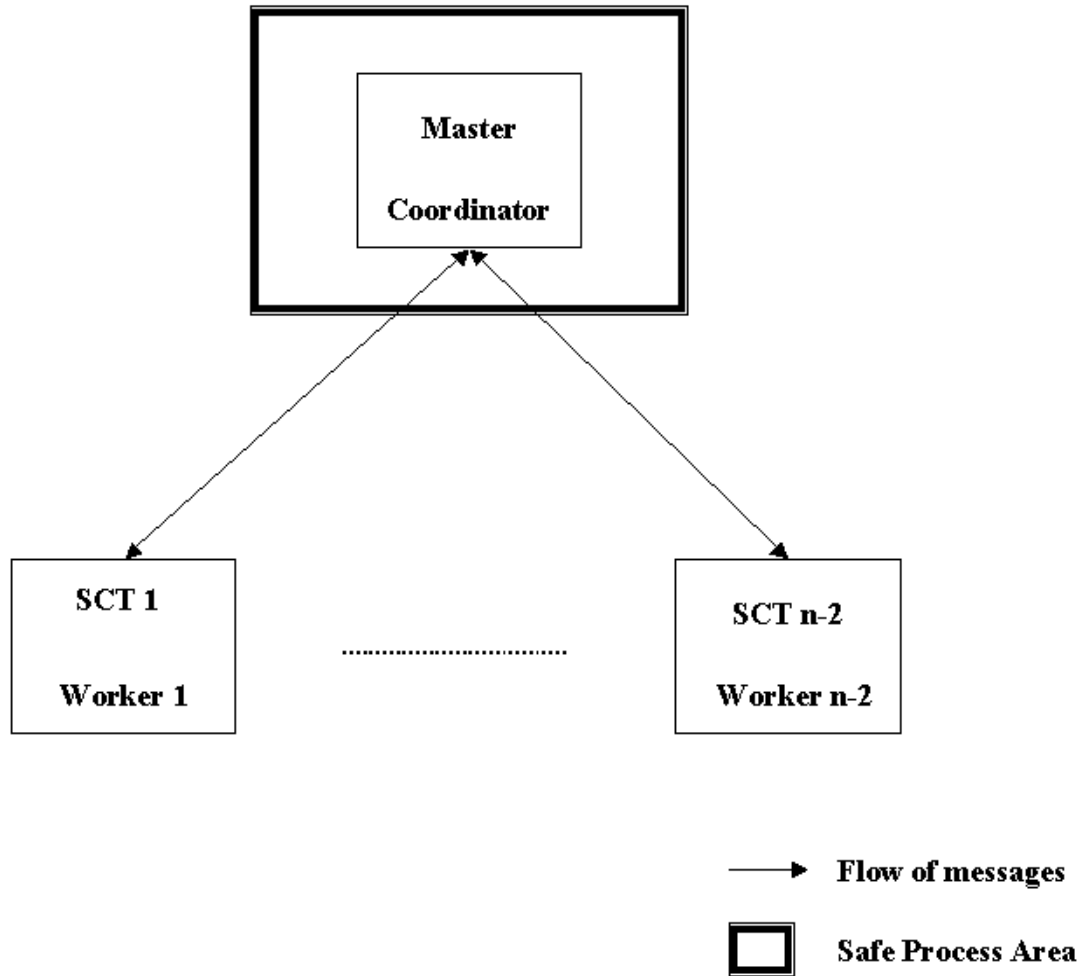


Figure 3.2: Model-Ia: Maser-Worker Application Model-Ia

Features:

This model requires that the middleware Coordinator detect the death of a worker process through the SCT. Detection and notification of faults in workers utilizing Coordinator and SCT is explained in Chapter 4. Middleware also provides services for the user-level recovery of a dead worker process. Checkpointing is not required in this model, as workers do not hold much state information and master is guaranteed against failures.

Applications:

Example applications of this model include the parallel message-passing version of Mandelbrot set visualization program [23], the Pmandel program, and the ray tracing applications for parallel image rendering. Experiments described in subsequent chapters will be performed using the Pmandel program.

3.3.2 *Model-IIa*

The all-interacting SPMD model is typically used in scientific applications and consists of a virtual all-to-all topology. Figure 3.3 presents this model.

Assumptions:

The model is more complex than the previous Master-Worker model. The assumption regarding the safe process area holds for this model and thus the rank 0 process, which equates to master process in Model-Ia and the coordinator thread, cannot crash or die. The processes are connected by a virtual all-to-all topology. Applications belonging to this model typically operate in an iterative loop. Each loop is marked by an exchange of messages with other processes followed by a computation phase or vice versa. These messages among processes are typically used for data exchange and are bounded by synchronization methods. Similar to Model-Ia, current implementation of Model-IIa cannot tolerate death of Rank 0. AEMs Model-IIb and Model-IIc are aimed at eliminating this problem through use of passive or active redundancy for Rank 0 and are currently under investigation.

Applications belonging to this model are assumed to be tightly coupled with regard to progress of entire application. Death of a single process in the application leads to stalling of the entire application. In the event of the death of a process, all MPI communication

between with the dead rank and collective communications must fail and return proper error codes. These error conditions will be used to drive processes into the recovery area.

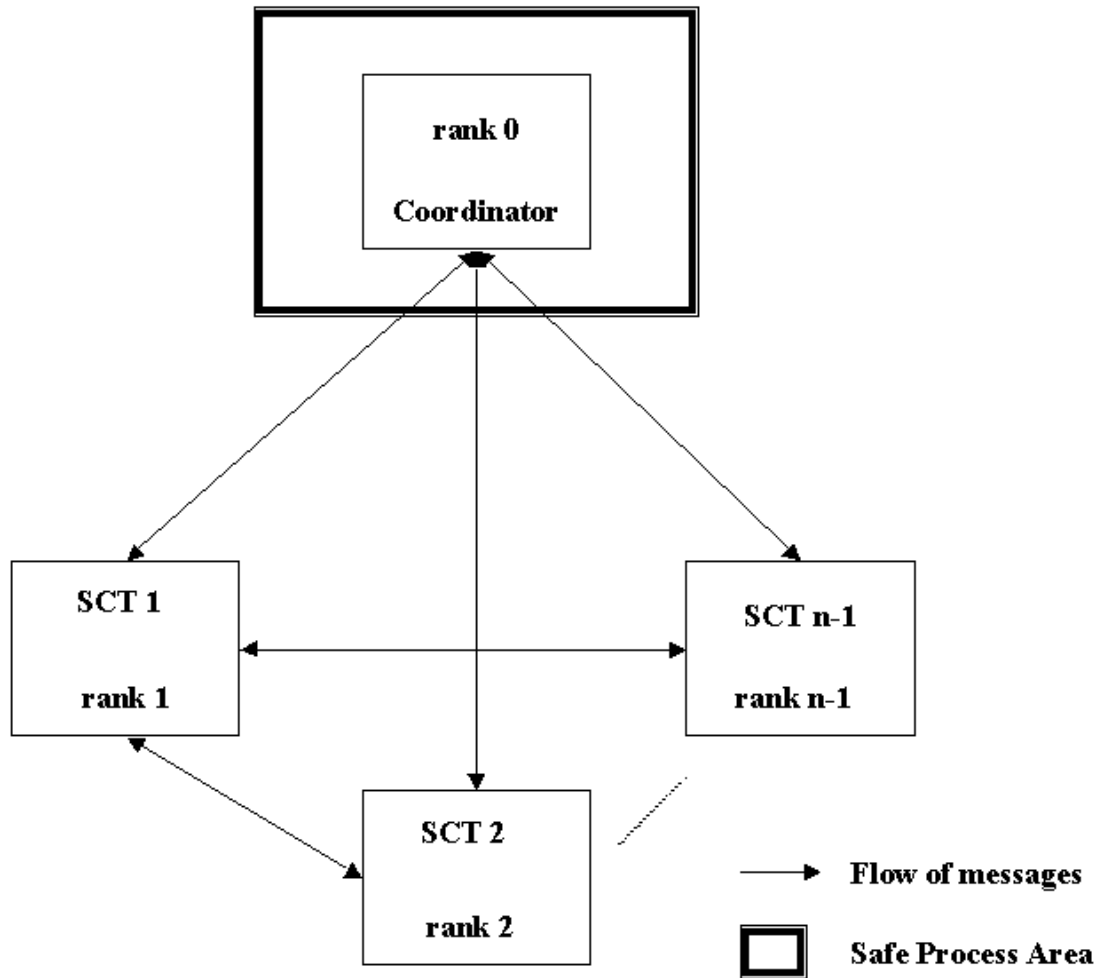


Figure 3.3: Model-IIa: SPMD all-interacting model

Features:

This model requires that the Coordinator detect the death of a process through the SCT heartbeats. The middleware also provides notification through dissemination of dead rank information to other alive ranks. Services also include user-level collective recovery of a dead rank and its proper induction into the process group. Since applications in this

model use an iterative loop, a choice must be presented through the API for a user-aware checkpointing. This API should allow users to define a data block to be checkpointed and complementary functions to retrieve checkpointed data.

Applications:

Example applications of this model typically include parallel discrete event simulation applications. Experiments will be conducted using the message-passing version of the Game of Life problem [6].

Chapter IV

APPROACH

The hypothesis is verified by developing MPI/FT [4]. MPI/FT is derived by incorporation of select fault-tolerant features into MPI/Pro [26], a high performance and multi-threaded implementation of MPI 1.1 standard [24]. This chapter describes the research methodology behind the design and development of MPI/FT. Apart from the modified middleware, applications also need to be modified to utilize MPI/FT API to make the applications fault-tolerant. These modifications, necessary in user applications to achieve fault-tolerance, are also described.

4.1 Research Methodology

The research methodology for this thesis can be broken down into five different activities. Applications refer to parallel programs developed with MPI. These applications are expected to complete successfully in the absence of external faults and are assumed to be devoid of internal design and implementation errors. The five activities of the research methodology are:

1. Identifying AEMs based on application characteristics features and fault-tolerance requirements.
2. Identifying select features to be incorporated for a given AEM (Design of MPI/FT).

3. Incorporating identified features to obtain MPI/FT. Implementation yields a middleware with fault-tolerance API (Development of MPI/FT).
4. Modifying existing applications to utilize a fault-tolerant API on the new middleware.
5. Studying and experimenting with unmodified and modified applications to obtain fault-free and fault-injected parameters. These parameters will be utilized to validate the hypothesis.

The first activity has been realized in Chapter 3. Subsequent sections in this chapter describe the next three activities. Parameters to determine overheads with MPI/FT and experiments to obtain those parameters are described in Chapter 5.

4.2 Usage

This section and its subsections describe MPI/FT from an application developer/user perspective. Achieving fault-tolerance requires fault detection, notification and recovery. The design of these steps for both Model-Ia and Model-IIa is described.

4.2.1 User level changes

Achieving fault-tolerance with MPI/FT requires the applications to be modified. These modifications are mainly targeted at utilizing MPI/FT API for fault notification and subsequent recovery procedures. A qualitative goal of MPI/FT is to provide a simple and powerful API.

4.2.1.1 Development and steps

The following are a list of steps a user would typically perform for an unmodified application.

1. Understand the problem and identify the parallelism in the program.
2. Decide on a program process model (master/worker, SPMD, hybrid, etc.) with determined communication patterns.
3. Implement (code) using a middleware.
4. Launch the application with the required number of processes. In the absence of external faults the application will successfully complete. A typical program launch could be

```
$ > mpirun -np 4 -mach_file myMachfile myParallelapp param1 param2
```

With the introduction of MPI/FT and its API, these user steps would be extended:

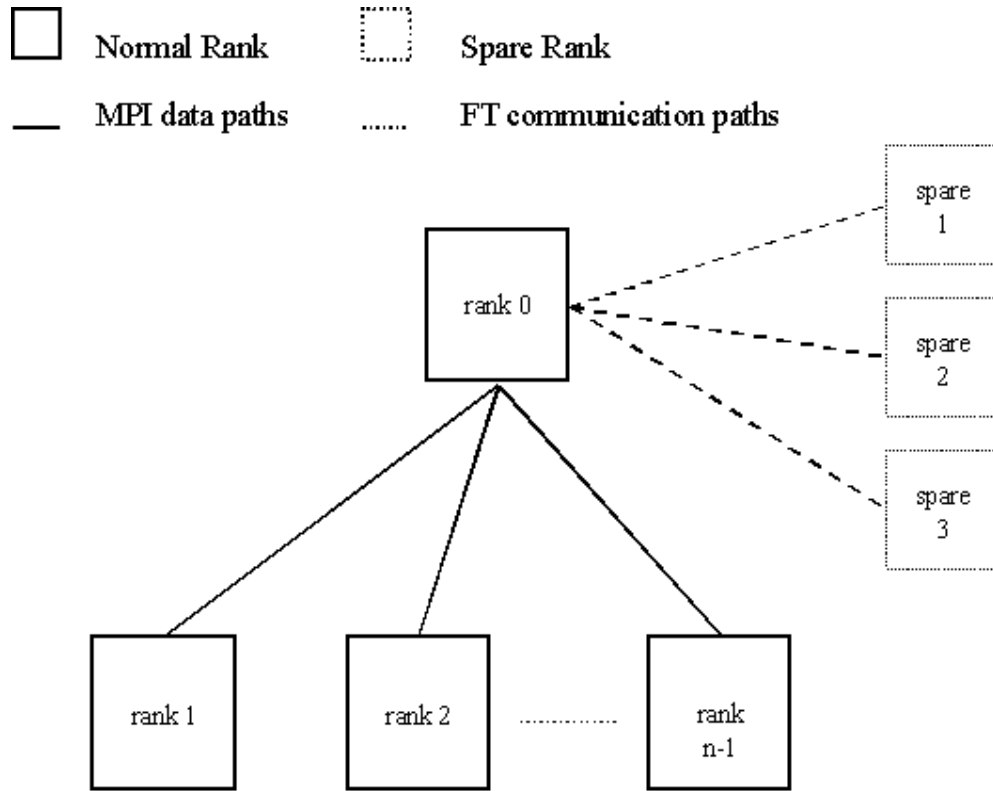
1. Understand the problem and identify the parallelism in the program.
2. Decide on a program process model (master/worker, SPMD, hybrid, etc.) with determined communication patterns.
3. Identify impacts of faults and regions in code for notification and recovery.
4. Decide on an AEM.
5. Implement (code) on a middleware and introduce FT-specific code.

6. Launch the application with required number of processes. Additionally, static spare processes should be launched. A typical program launch with fault-tolerance could be

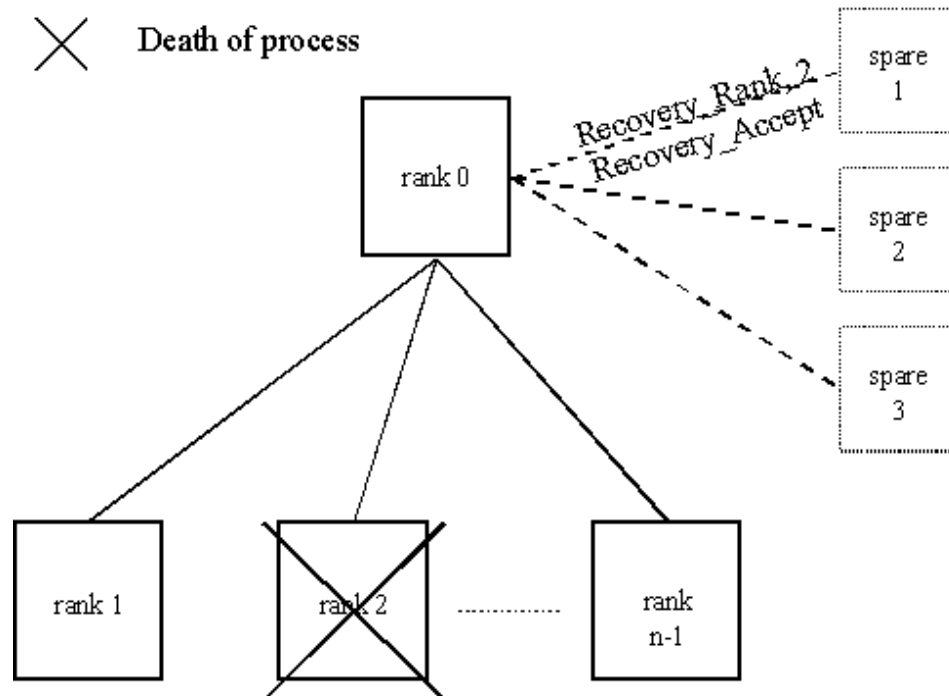
```
$ > mpiftrun -np 4 -sp 2 -mach_file myMachfile -ftparam1 val1 -ftparam2 val2  
myFTParallelapp param1 param2
```

The modified program launcher passes MPI/FT-specific parameters to applications. Currently these FT parameters can be utilized to specify values for controlling internal and external heartbeat frequency and their timeouts.

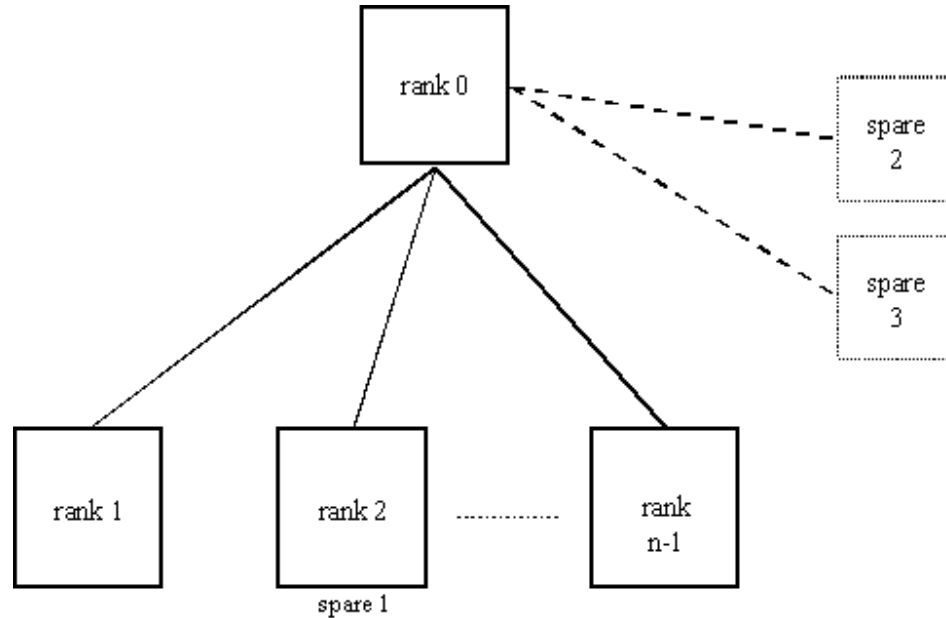
The "sp" option allows users to specify the number of spare ranks in anticipation of faults and process deaths. These spare ranks are hibernated until required and later assume a new rank as directed by the coordinator. Figure 4.1 presents these cases. Dynamic allocation of spare processes may be possible with integration of MPI/FT and a cluster manager or scheduler.



a) Spare Ranks in Hibernation, Normal Run



b) Spare Rank Release and Recovery



c) Spare Rank joins Normal Run, After Recovery

Figure 4.1: Utilization of Spare Ranks in MPI/FT

4.2.1.2 FT API and code modifications

This section introduces and describes MPI/FT API. Example pseudo-code utilizing the FT API is also shown for each model.

4.2.1.2.1 Model-Ia

The following is the MPI/FT API available for Model-Ia (simple master/worker).

1) Get Dead Rank information

```
int
MPIFT_GetDeadRanks (
    OUT int *deadcount,
    OUT int *deadarrayranks,
    IN int array_size)
```

This function returns the information about ranks considered as dead by the detection process. Information (count, actual dead rank number) returned by this function will be used in initiating recovery. It should only be invoked from the master/ rank 0 process. Detection of death of a rank and actual notification procedures are explained in Section 4.3. Currently, this is the only API available for notification of a dead rank.

2) Recover a dead worker/rank

```
int
MPIFT_RecoverRank(
    IN int RankToRecover
);
```

This function initiates recovery of a dead worker/rank. It should only be invoked from the master/ rank 0 process. Actual working of recovery process is explained in Section 4.3.

Pseudo-code for Model-Ia applications is presented in Figures 4.2 and 4.3. Figure 4.2 presents code for a sample master/worker application. Figure 4.3 presents one way of making the program fault-tolerant by use of MPI/FT API. Note that the amount of MPI/FT API usage may remain constant irrespective of the actual normal code size in this case.

```

/* Plain version : Master/worker */
main(..){
    /* variables defined here*/
    int rank, size, *main_area[][];

    /* MPI_Init */
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    /* Initial parameters */
    if (rank ==0 ){
        ..
        MPI_Send(init_data, ..);
    }
    else{
        ..
        MPI_Recv(init_data, ..);
    }

    if(rank ==0){
        create_jobs(jobs_array);
        for ( I=1..n-1)
            MPI_Send (intial_jobs);
            Job_array --;
    };

    // actual work loop
    while( job_array != NULL){

        MPI_Recv( result,.. fromanyworker,..);
        ..
        results_array [] = result;
        MPI_Send( job_array,.., tolastworker .. );
        jobs_array --;

    };

    if(rank ==0){
        Write( file);
    };
    MPI_Finalize();
}

```

Figure 4.2: Pseudo-code for Unmodified Model-Ia Application

```

/* FT version : Master/worker */
main(..){
    /* variables defined here*/
    int rank, size, *main_area[][];

    /* MPI_Init */
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    /* Initial parameters */
    if (rank ==0 ){
        ..
        MPI_Send(init_data, ..);
    }
    else{
        ..
        MPI_Recv(init_data, ..);
    }

    if(rank ==0){
        create_jobs(jobs_array);
        for ( I=1..n-1)
            MPI_Send (initial_jobs);
            Job_array --;
            last_job_worker[I] = initial_job;
    };

    // actual work loop
    while( job_array != NULL){
        MPIFT_GetDeadRanks (&deadrankarray,deadcount);

        if( deadcount >0){
            for( counter = 0 .. deadcount-1)
                deadrank = deadrankarray[counter];
                // Retrieve job
                jobs_array [] = last_job_worker [deadrank];
                jobs_array++;
                // Recover Rank
                MPIFT_RecoverRank (deadrank) ;
                // Initialize and assign a new job
                MPI_Send(init_data,..,deadrank,..);
                MPI_Send (initial_jobs);
                Job_array --;
                last_job_worker[deadrank] = initial_job;
            };
            MPI_Recv( result,.. fromanyworker,..);
            ..

```

Figure 4.3: Pseudo-code for Modified Model-Ia Application

```

results_array [] = result;
    MPI_Send( job_array,..., tolastworker .. );
    jobs_array --;

    // Worker might die, save job
    last_job_worker[tolastworker] = job_array[];

};

if(rank ==0){
    Write( file);
};
MPI_Finalize();
}

```

Figure 4.3 (Continued): Pseudo-code for Modified Model-Ia Application

4.2.1.2.2 Model-IIa

The following is the MPI/FT API available for Model-IIa (SPMD). Model-IIa applications are marked by a similarity in functionality at all processes. Moreover, Model-IIa applications are expected to run in a tightly coupled manner, and death of a single process in the application stalls all processes. This tight coupling and similarity translates to this FT API being called in a symmetrical manner from all ranks/processes. Model-IIa also provides API for user-aware checkpointing. Users/developers are expected to marshal data and application state information to be stored by the checkpointing routines. These routines return this marshaled data during recovery for application state rollback and recovery.

1) Get Dead Rank information

Syntax and semantics for this function are same as in Model-Ia. Unlike in Model-Ia, this function is invoked at all ranks/processes.

2) Recover a dead rank

Syntax and semantics for this function are same as in Model-Ia. Unlike in Model-Ia, this function must be invoked at all ranks for successful recovery.

3) Initiate Checkpoint

```
int
MPIFT_ChkptDo(
    IN void      *data_to_store,
    IN int       data_size,
    OUT int      *chkpt_num,
    IN MPI_COMM communicator
);
```

This function takes in data/state information provided by the user and stores them for later retrieval. This function call is a collective call and hence should be invoked by all the ranks/processes in a communicator. Checkpoints are stored in files, and all checkpoints stored in a single call share common identification number for later retrieval. For Model-IIa applications, it is expected that this function will be invoked with the default global communicator, the MPI_COMM_WORLD.

4) Recover Checkpoint data

```
int
MPIFT_ChkptRecover(
    OUT void      *data_retrived,
    IN int        in_data_size,
    OUT int       *out_data_size,
    OUT int       *chkpt_num_retrieved,
    IN MPI_COMM  communicator
);
```

This function should be invoked after a dead rank/process is recovered at the middleware level. Middleware decides on the latest and complete checkpoint number valid at each process for a given communicator. After agreement, the data

associated with that checkpoint number is available to users. Users should use this retrieved data to rollback to an agreeable previous application state. This function must be called by all processes in the communicator and will fail in the presence of dead processes.

Pseudo-code for Model-IIa applications is presented in Figures 4.4 and 4.5. Figure 4.4 presents pseudo-code for the Game of Life [6] application. Figure 4.5 presents one way of making the program fault-tolerant by use of MPI/FT API. The data that is marshaled for checkpointing is the iteration counter value and data region of each individual process.

```

/* Plain version : Game of life */
main(..){
    /* variables defined here*/
    int rank, size, *main_area[][];

    /* MPI_Init */

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);

    /* Initialized data and parameters */
    proc_rows = atoi (argv[1]);
    ..

    /* Initial data distribution */
    if (rank ==0 ){
        filesdes = open (input_filename, O_RDONLY);
        for( i=1; i<size; i++){
            ..
            MPI_Send(init_data, ..);
        };
    }
    else{
        //recv init data
        ..
        MPI_Recv(recv_buf, ..);
    }

    // actual work loop
    while(i <max_iterations){

        // communication: exchange data with neighbours
        MPI_Isend( (main_area0[1] + 1), .. );
        ..
        ..
        MPI_Irecv( (main_area0[1] + 1), .. );

        // computation: Evaluate next state
        for(j=1; j<my_data_rows+1; j++){
            for(k=1; k<my_data_cols+1; k++){
                //For each cell compute next state
                // using game of life logic
            };
        };
        MPI_Barrier(MPI_COMM_WORLD);
        i++;
    };
};

```

Figure 4.4: Pseudo-code for Unmodified Model Ila Application


```
    end = MPI_Wtime();  
    // Program end , collect data  
    if(rank ==0){  
        //collect data from all ranks  
        MPI_Recv (..);  
        Write( file);  
    }else{  
        MPI_Send(final_result, .., .., .., .., .., ..);  
    };  
    //MPI end  
    MPI_Finalize();  
}
```

Figure 4.4 (Continued): Pseudo-code for Unmodified Model-IIa Application

```

/* FT version : Game of life */
main(..){
    /* variables defined here*/
    int rank, size, *main_area[][];

    /* MPI_Init */

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    /* Initialized data and parameters */
    proc_rows = atoi (argv[1]);
    ..
    /* Initial data distribution */
    if (rank ==0 ){
        filesdes = open (input_filename, O_RDONLY);
        for( i=1; i<size; i++){
            ..
            MPI_Send(init_data, ..);
        };
    }
    else{
        //recv init data
        ..
        MPI_Recv(recv_buf, ..);
    }
    // actual work loop
    while(i <max_iterations){

        MPIFT_RecoveryPoint();

        MPIFT_GetDeadRanks(&deadcount, deadranks);
        if( deadcount >0 ){
            /*
            for(counter =0 ; counter <deadcount; counter++){
                // recover deadrank
                MPIFT_RecoverRank (deadranks [counter] );
            }

            // new spare init
            MPIFT_ChkptRecover (&chkpt_buf, ..);
            //unmarshall data
            memcpy(to_ptr, chkpt_buf, ..);
        };
    }

```

Figure 4.5: Pseudo-code for Modified Model-IIa Application

```

// communication: exchange data with neighbours

MPI_Isend( (main_area0[1] + 1), .. );
..
..
MPI_Irecv( (main_area0[1] + 1), .. );

// computation: Evaluate next state
for(j=1; j<my_data_rows+1; j++){
    for(k=1; k<my_data_cols+1; k++){
        //For each cell compute next state
        // using game of life logic
    };
};
MPI_Barrier(MPI_COMM_WORLD);

// Marshall data for checkpoint
memcpy(chkpt_buf, from_buf, ..);
MPIFT_ChkptDo(&chkpt_buf, ..);
i++;
};
end = MPI_Wtime();

// Program end , collect data

if(rank ==0){
    //collect data from all ranks
    MPI_Recv (..);
    Write( file);

else{
    MPI_Send(final_result, .., .., .., .., ..);
};
//MPI end
MPI_Finalize();
}

```

Figure 4.5 (Continued): Pseudo-code for Modified Model-IIa Application

4.3 Design and Implementation

Fault-tolerance is achieved in MPI/FT by following fundamental steps of fault detection, notification, and recovery. These steps are presented in Figure 4.6. Each step is essential and is explained in the following subsections.

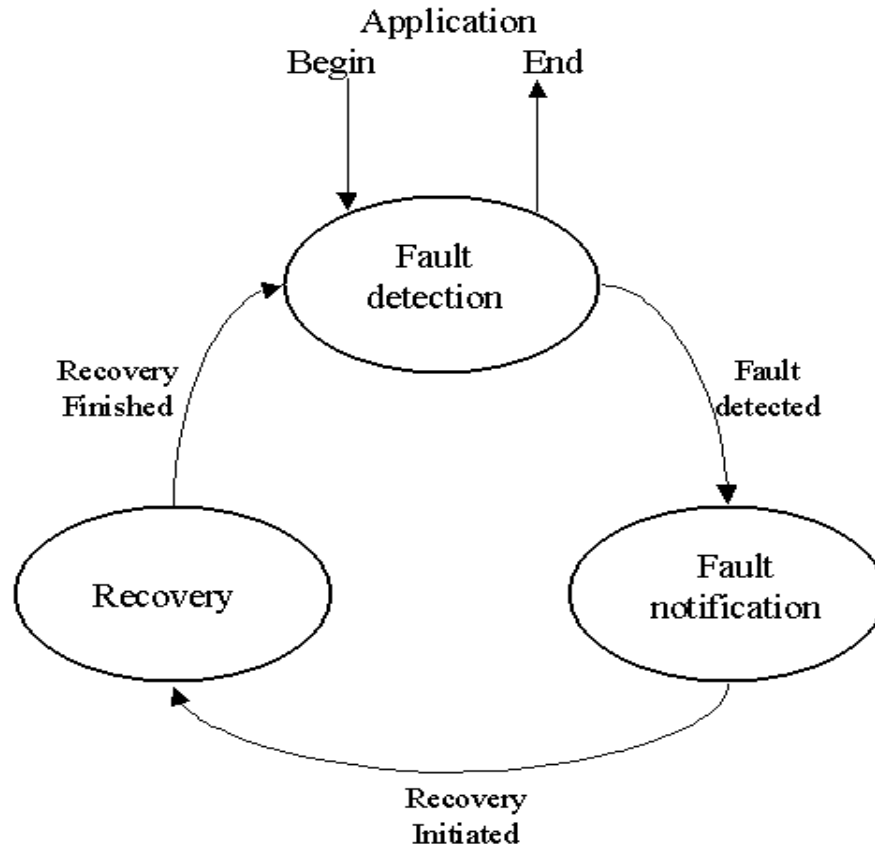


Figure 4.6: Primary Steps in Achieving Fault-tolerance

4.3.1 Detection

Fault-detection is an important step in achieving fault-tolerance. A good fault-detection strategy must be accurate (minimize false positives), fast (quick detection between actual fault and detection) and cheap (impose low overheads). Since fault-detection strategy is typically the main contributing component towards fault-free overhead, a fault-detection strategy should be an appropriate compromise among its various features.

The fault-model assumed for the implementation of MPI/FT consists primarily of fail-stop process deaths and hanging or misbehaving progress threads. Loss of communication channels is also associated with the death of processes. Detection of

anomalies in a user thread is currently not supported, but this may be supported in the future with either explicit user-assisted detection or implicit heartbeats. Additionally transient faults that lead to production of incorrect results but allow MPI application to continue are not covered. Detection of such transient errors must be provided by the application through explicit use of ABFT (Application Based Fault Tolerance) [18, 31]

Detection, notification and recovery are all achieved through the additional FT thread introduced in each process. This FT thread is named Coordinator at master/rank 0 to signify its role in detection and recovery. The FT thread is named SCT at other processes. SCT is a powerful concept with varying levels of portability and functionality [4].

- 1) Trivially non-portable: Uses existing internal data structures of a particular MPI implementation and performs trivial/obvious checks on them. It is not visible to the user.
- 2) Trivially portable:
 - (a) Uses the PMPI profiling interface provided by the MPI standard to extend the previous approach across all MPI implementations. The complexity of operations that can be achieved is still trivial. It is visible to the user.
 - (b) Adheres to the specifications of TotalView [12] and provides access to MPI internal structures across all MPI implementations. These structures include the send and receive queue. It is visible to the user.
- 3) Non-trivially portable: Extends the functionality in (1) and (2) by incorporating intelligence into the consistency checks. Can be visible to the user.

4) Non-trivially non-portable: This approach provides the most general functionality by defining new internal structures to aid consistency checks, etc. Such structures and checks are specific to an MPI implementation.

The current MPI/FT's SCT implementation operates at level 1, trivially non-portable, and is primarily used for detection through heartbeats. Heartbeats are used both externally and internally.

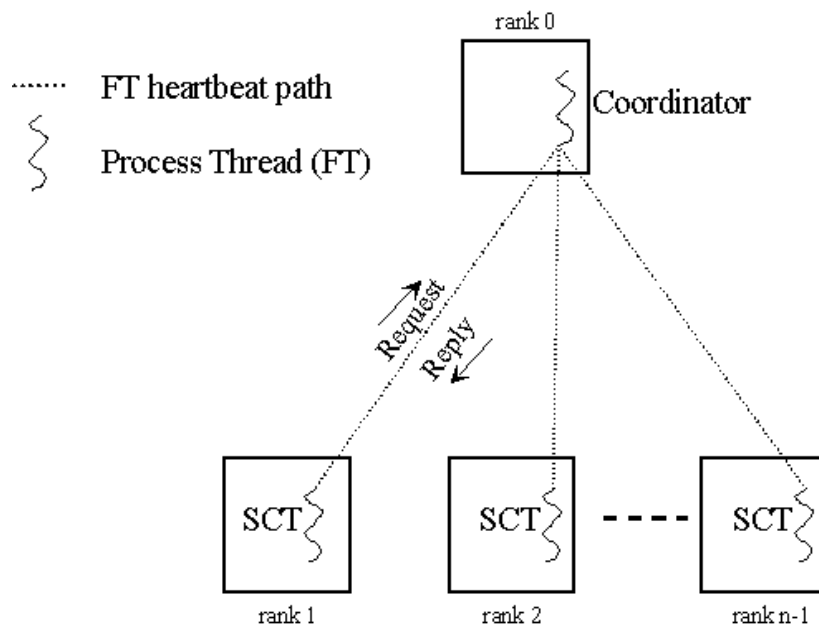


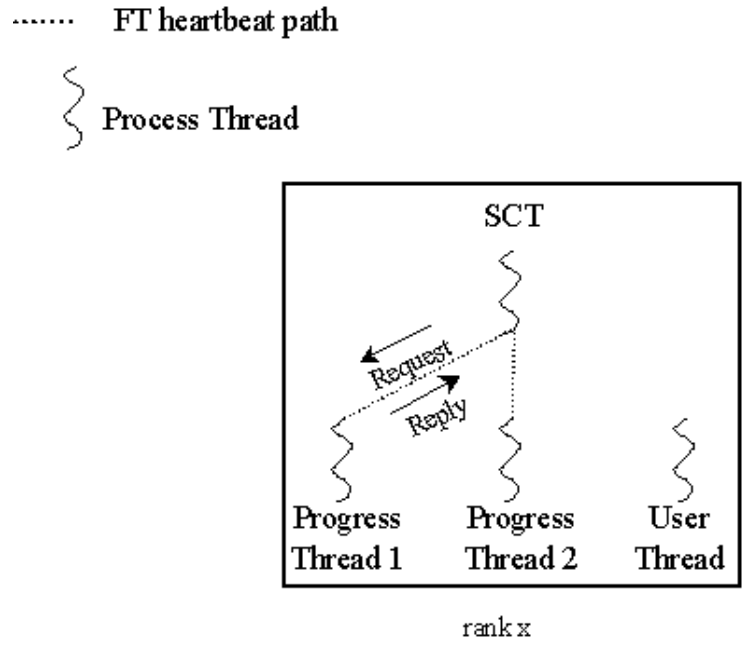
Figure 4.7: Coordinator and SCT: External heartbeat mechanism

External heartbeats are used by Coordinator to detect process deaths. Coordinator passively listens to periodic alive messages from SCTs of other ranks. If an alive message is not received in a certain timeout, Coordinator determines that the unresponsive rank is dead. Coordinator also determines death of a rank if the connections to that process are

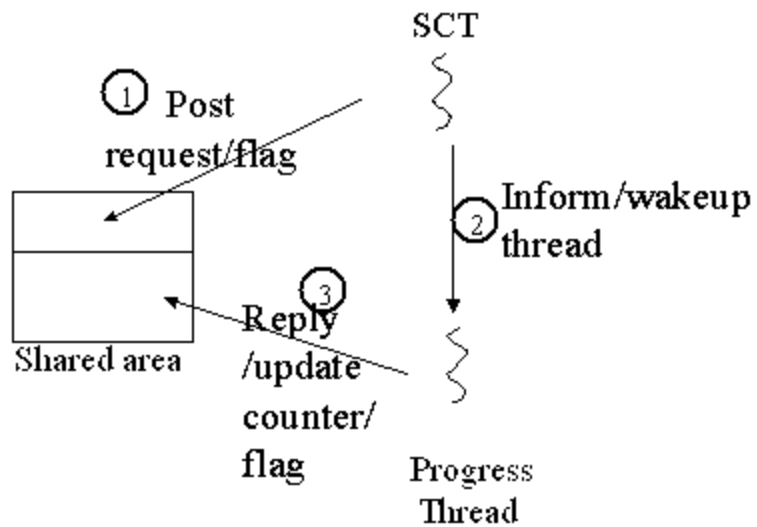
abnormally terminated. In either case, the user thread of the application is appropriately notified. Figure 4.7 presents the interaction of processes for external heartbeat.

SCT, apart from sending periodic alive messages to Coordinator, performs internal checking of other progress threads. MPI/Pro, the base implementation of MPI/FT, consists of progress threads. These progress threads are critical for actual message-passing operations. SCT actively requests these threads for status information. SCT and progress threads interact through a shared memory area. SCT posts a request for reply and notifies the progress thread. Upon receiving the notification, progress threads reply through these shared areas. If a progress thread does not reply to a posted request within a user-configurable interval, SCT decides that the progress thread is corrupted or has crashed and informs the Coordinator. Coordinator acknowledges this information and sends commands instructing the SCT to terminate the rank. Figure 4.8 captures interactions between SCT and other progress threads. Coordinator proceeds to notification and appropriate recovery measures.

It is possible that while processing large communication requests, progress threads will not be able to respond to an SCT request in time, thereby causing false failure notifications. In order to prevent this situation, progress threads can inform SCT of the actual start and expected end time of an operation. During this busy time, SCT does not post new status requests for that progress thread.



(a) SCT and Progress Threads



(b) Typical Interaction

Figure 4.8: SCT and Progress Threads: Internal heartbeat mechanism

Model-Ia and Model-IIa both have similar detection mechanisms. However, in Model-IIa the flow of heartbeat requests and replies between Coordinator and SCT is

reversed. Coordinator, instead of passively hearing alive messages, actively sends out request messages to which ranks reply. This reversal is essential, as the scope of notification in Model-IIa is different than in Model-Ia. Both external and internal heartbeat rates and their timeout values are user configurable and can be passed through the command line to the application launcher (mpiftrun).

4.3.2 Notification and Recovery

Notification deals with dissemination of information relevant to fault-detection, while recovery consists of actual steps to mask and recover from the fault.

4.3.2.1 Model-Ia Recovery

Model-Ia consists of simple master/worker applications where death of a worker will only affect the master process or rank. Hence, only the master process is informed of the death while other worker processes are unaware of the death. Death of a worker process or rank can be tolerated by replacing it with a passive spare process. Recovery of a worker process is meaningful only in the context of the Coordinator. The virtual star-topology assumed for this model precludes connection of the new spare/worker to the rest of the workers.

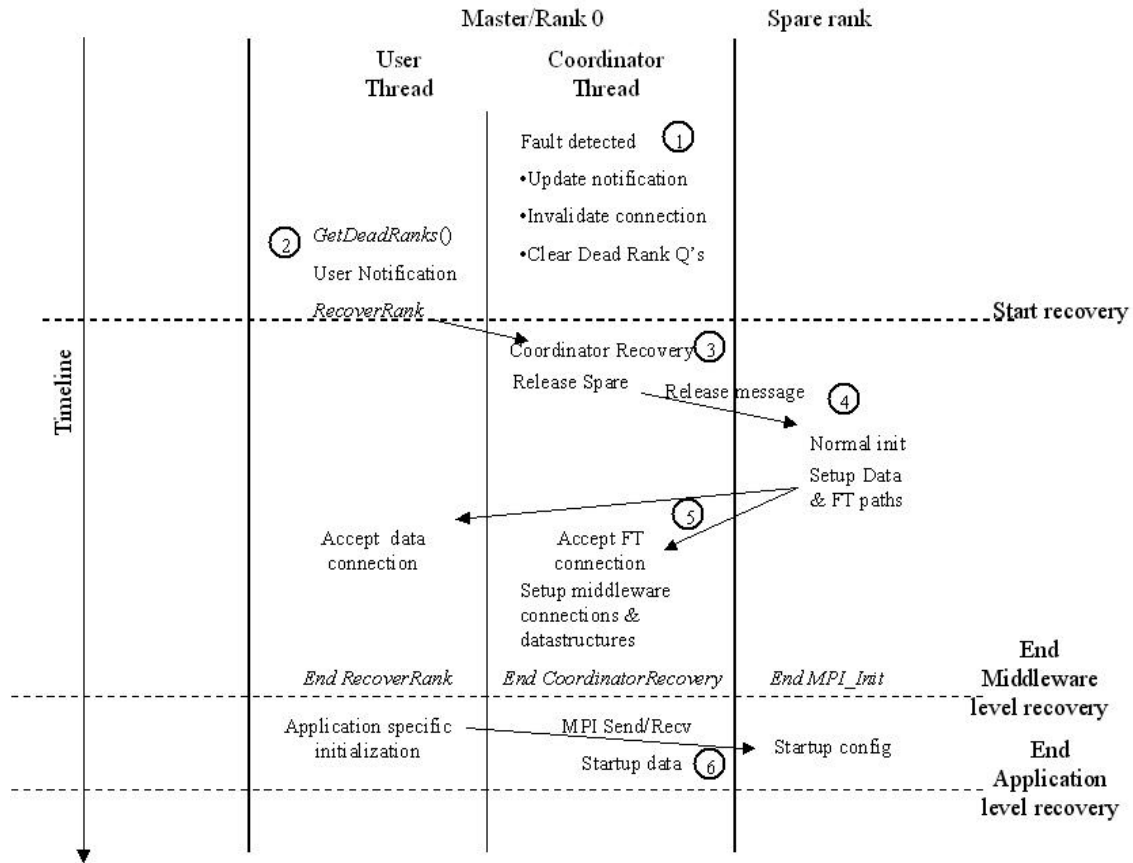


Figure 4.9: Notification and Recovery in Model-Ia

The following are the steps involved during recovery in Model-Ia. Figure 4.9 presents these events along a timeline. Steps followed in both the user thread and the Coordinator at the master process are shown.

1. Coordinator detects the death of a worker, updates the notification area, and invalidates connections to the dead rank. Any MPI calls to this dead rank are not initiated and return with errors. Internal queues for this rank are cleared.

2. The user thread retrieves dead rank information by calling `MPIFT_GetDeadRanks()`. The user thread initiates recovery by calling `MPIFT_RecoveryRank()`.
3. `MPIFT_RecoveryRank()` initiates Coordinator recovery steps.
4. Coordinator releases the spare rank with the dead rank number.
5. Both the user thread and Coordinator wait for the new spare to connect. Reconnection of the spare marks the end of middleware recovery.
6. The user thread proceeds with application level recovery and state initialization.

4.3.2.2 Model-IIa Recovery

Model-IIa consists of SPMD all-interacting processes. These kinds of applications cannot proceed even when a single process/rank is dead. Thus, information about a dead rank must be available to the alive ranks. Coordinator, upon detecting a dead rank, disburses this information to all alive ranks. SCTs will utilize this information to set the information about dead ranks.

While most of the recovery steps for rank 0 and other alive ranks are the same, there are differences during notification and initial recovery steps. This difference is expected, as the Coordinator needs to drive the rest of the ranks to recovery. Figure 4.10 summarizes these recovery steps.

The following are the recovery steps at the Coordinator/rank 0:

1. Send notification to SCT of other alive ranks.

2. The user thread retrieves the dead rank information by calling `MPIFT_GetDeadRanks()`. The user thread initiates recovery by calling `MPIFT_RecoveryRank()`.
3. `MPIFT_RecoveryRank()` initiates recovery steps at the Coordinator.
4. All message queues are cleared.
5. Release suspended spare process with recovery rank.
6. Accept data and FT connections from the new spare.
7. Start application level recovery through `MPIFT_ChkptRecover()` and later unmarshal and restore application state.

The following steps occur for checkpointing recovery (step 7).

1. Agree on a lowest valid checkpointing number amongst all processes. A valid checkpointing number for a process is one where a process can access all stored data without errors.
2. Retrieve information from the last checkpoint and return to the user.
3. Users later unmarshal this data and restore process state in an application-specific manner.

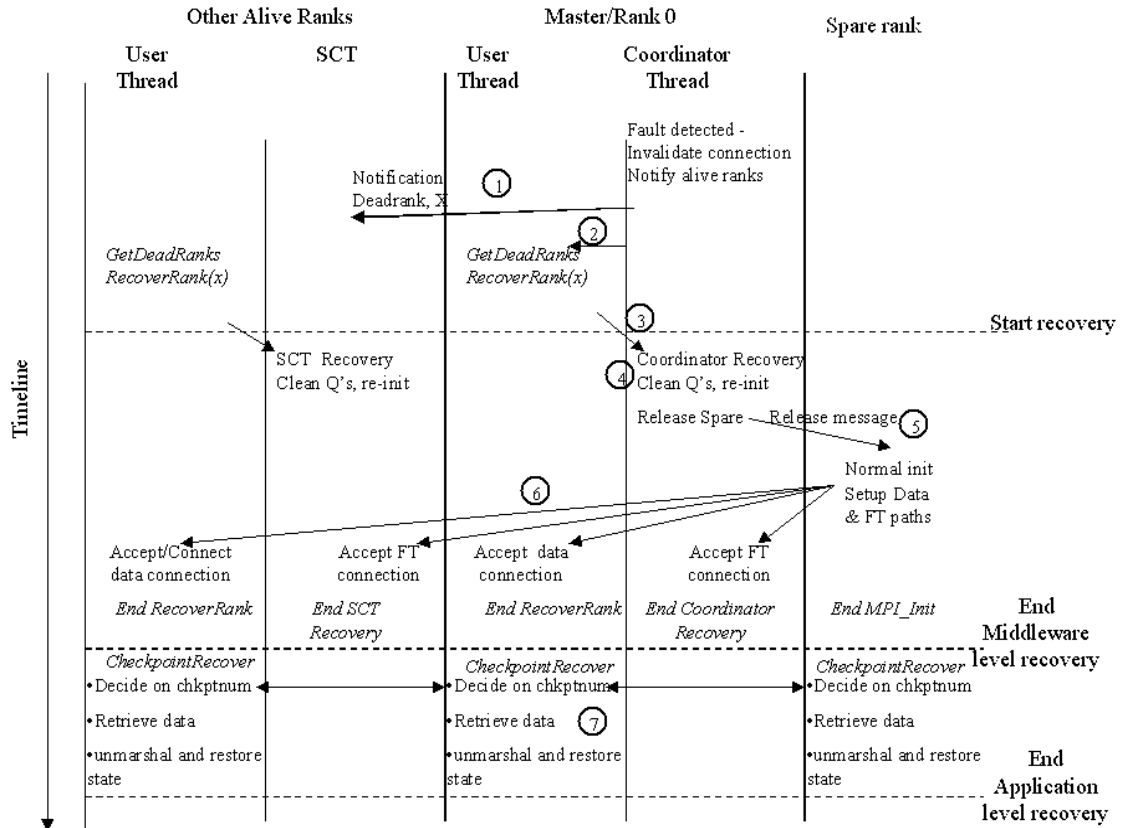


Figure 4.10: Notification and Recovery in Model-IIa

As mentioned earlier, all processes of the communicator must be alive both before and during the checkpoint recovery function. If a process is dead either before or during the checkpoint recovery process, application-level recovery will fail. Application-level recovery may be retried after middleware-level recovery of the dead process is completed. Thus, by a combination of middleware and application level recovery, Model-IIa applications can successfully recover from faults and progress to completion.

Chapter V

EXPERIMENTS, RESULTS, AND ANALYSIS

This chapter presents parameters, experiments, and results to validate the hypothesis. The hypothesis consists of two parts that focus on successful completion in the presence of modeled faults and acceptable overheads in the absence of such faults. Parameters are separated into fault-free and fault-injected categories. Experiments and results from fault-injected category are focused on proving that applications complete successfully in the presence of faults. Fault-injected parameters capture the recovery time both at the middleware and application level.

Experiments and results from fault-free category are focused on proving that fault-free overheads are acceptable in the absence of faults. Fault-free overheads are defined as the additional costs incurred by the application in the absence of faults. Fault-detection mechanisms and additional masking measures during message-passing are the primary contributors to fault-free overheads. Fault-free overheads are measured for message-passing, run-time of applications, and changes made to programs.

Experiments were conducted using a combination of standard and custom defined programs. Message-passing overheads were measured by a ping-pong program. Fault-free and fault-injected overheads were measured by using sample applications for both

Model-Ia and Model-IIa. Overheads, in cases applicable, were computed as the percentage change in parameters obtained from MPI/FT [4] and MPI/Pro [26]. Such a computation is justified as MPI/FT has been realized by selective incorporation of fault-tolerant features into MPI/Pro. The following section describes experiments and measurements for evaluating message-passing overheads. Subsequent sections present experiments and results for both Model-Ia and Model-IIa.

5.1 Message-passing Overheads

Latency and bandwidth are primary parameters for evaluating the performance of a message-passing system. Latency refers to the delay in sending the message between two MPI processes of an application. Factors contributing towards latency are the physical characteristics of the network and processing of messages at nodes. Bandwidth refers to the effective throughput between two MPI processes of an application. Bandwidth is calculated as the ratio between message size and time taken to transfer.

Latency and bandwidth were obtained using a ping-pong program for various message sizes. As explained in Chapter 4, fault-detection in MPI/FT is primarily performed by a combination of external and internal heartbeats. Measurements for latency and bandwidth were obtained at different rates of external and internal heartbeats. Model-Ia and Model-IIa, as explained in Section 4.3.1, employ similar fault detection mechanisms, and hence message-passing overheads for both these models are similar. Therefore only results from Model-Ia are presented for studying message-passing overheads.

5.1.1 Latency Overhead

Fault-free overhead of latency is defined as the percentage increase in latency time in the absence of faults because of introduction of fault-tolerance mechanisms. The definition is given by equation 5.1,

$$Overhead_{Latency} = \left(\frac{T_{MPI/FT} - T_{MPI/Pro}}{T_{MPI/Pro}} \right) \times 100, \quad (5.1)$$

where $T_{MPI/FT}$ is the latency for a message on MPI/FT and $T_{MPI/Pro}$ is the latency for a message on MPI/Pro middleware.

5.1.1.1 Experimental Setup

Figure 5.1 presents the pseudo-code of the ping-pong program for obtaining one-way latency. This one-way latency is obtained by calculating half of the measured roundtrip time required to send and receive a single MPI message. Latency results were first collected by running this ping-pong test program compiled with plain MPI/Pro middleware. Results were also obtained from the ping-pong test program compiled with MPI/FT middleware. Fault-free overhead was then calculated as the percentage increase in MPI/FT case when compared to MPI/Pro results. These results were computed for message sizes ranging from 0 to 1 megabyte. Results were also computed for various rates of external and internal heartbeats. The ping-pong test was run on a cluster of Intel Pentium machines (750mhz, 512 MB RAM, Linux 2.4 OS).


```

if(rank == 0){
    MPI_Send_init(sendBuff, DATASIZE, ..);
    MPI_Recv_init(recvBuff, DATASIZE, ..);
    starttime = MPI_Wtime();
    for(j = 0; j < numTests; j++){
        MPI_Start(&sendReq);
        MPI_Start(&recvReq);
        MPI_Wait(&sendReq, &status);
        MPI_Wait(&recvReq, &status);
    }
    endtime= MPI_Wtime() ;
    time = (endtime-starttime)/numTests;
    latency = time / 2.0 / numTests;
}else{
    MPI_Send_init(recvBuff, dataSize, ..);
    MPI_Recv_init(recvBuff, dataSize, ..);
    for(j = 0; j < numTests; j++){
        MPI_Start(&recvReq);
        MPI_Wait(&recvReq, &status);
        MPI_Start(&sendReq);
        MPI_Wait(&sendReq, &status);
    }
}
}

```

Figure 5.1: Pseudo-code for measuring Latency

The introduction of fault-detection features and of additional checks in the path of message-passing was expected to increase latency. It was hypothesized that acceptable fault-free overhead would be 5%.

5.1.1.2 Results and Analysis

Figures 5.2 and 5.3 present the overhead in latency. Figure 5.2 presents the overhead with various values of external heartbeats, while internal heartbeats were disabled. Figure 5.3 presents overheads at various internal heartbeat rates while the external heartbeat rate is set at 0.25 Hz. These results are presented along a logarithmic axis to accommodate the large range of message sizes.

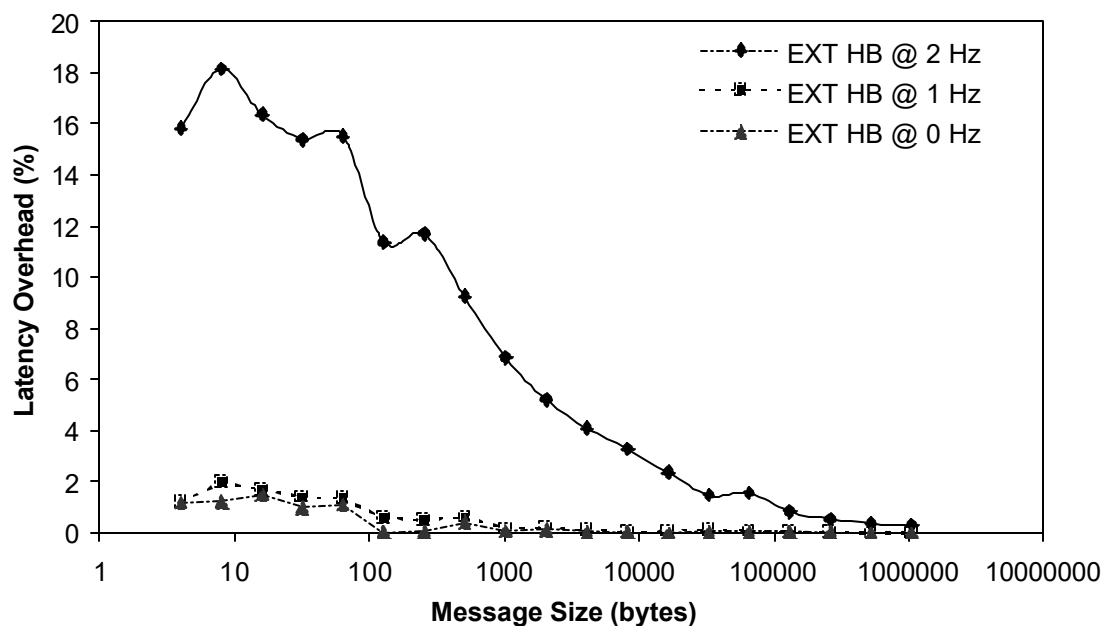


Figure 5.2: Latency Overheads with only External Heartbeats

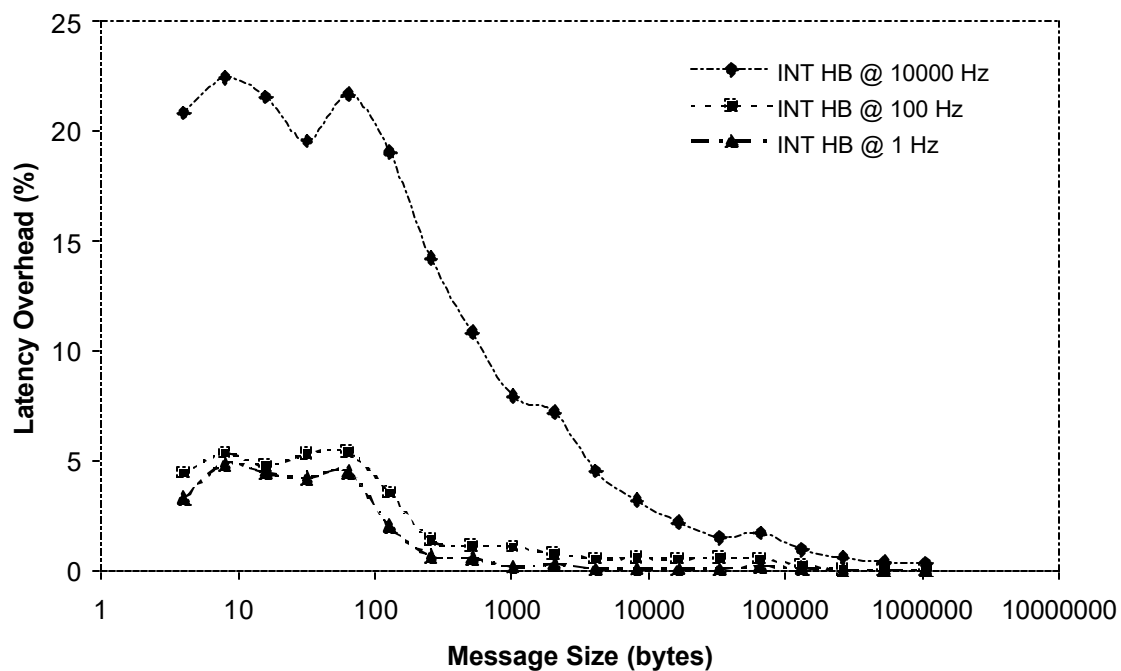


Figure 5.3: Latency Overheads with Internal and External Heartbeats

Figure 5.2 consists of latency overheads at zero-frequency of external heartbeats. These zero-frequency overheads indicate the impact of additional checks in the path of sending and receiving MPI messages. Overheads at higher frequencies indicate the combined impact of the additional checks and the impact of FT thread execution on original user and progress threads. Results indicate these increasing overheads with increasing rates of external and internal heartbeats. Latency overheads also decrease with increase in message size. This may be attributed to the decreasing impact of the constant amount of checks on each message and the busy time concept of SCT. The busy time concept, as described in Section 4.3.1, allows progress threads to inform SCT about duration of long tasks, during which SCT stops polling progress threads.

It may be noticed that the latency graphs from Figures 5.2 and 5.3 consist of spikes at certain message sizes, especially at lower message sizes. These spikes do not appear at the 32-kilobyte message size, where the message transfer protocol changes for small to large message sizes [26]. In order to explain these spikes Figure 5.4 presents latency of MPI/Pro and latency overheads for smaller message sizes. It is evident that many of the spikes present in the logarithmic graph are less prominent in the graph with the non-logarithmic axis. The only prominent spike exists at a message size of 8 bytes, and this spike may be attributed to an accompanying sudden decrease in MPI/Pro latency, while the amount of additional checks remain the same. In conclusion, latency overheads remain low at normal rates of heartbeats. In the case of long running programs with a total runtime on the order of days, a much slower rate of heartbeats can be utilized. Such low rates of heartbeats can have a negligible impact on latency.

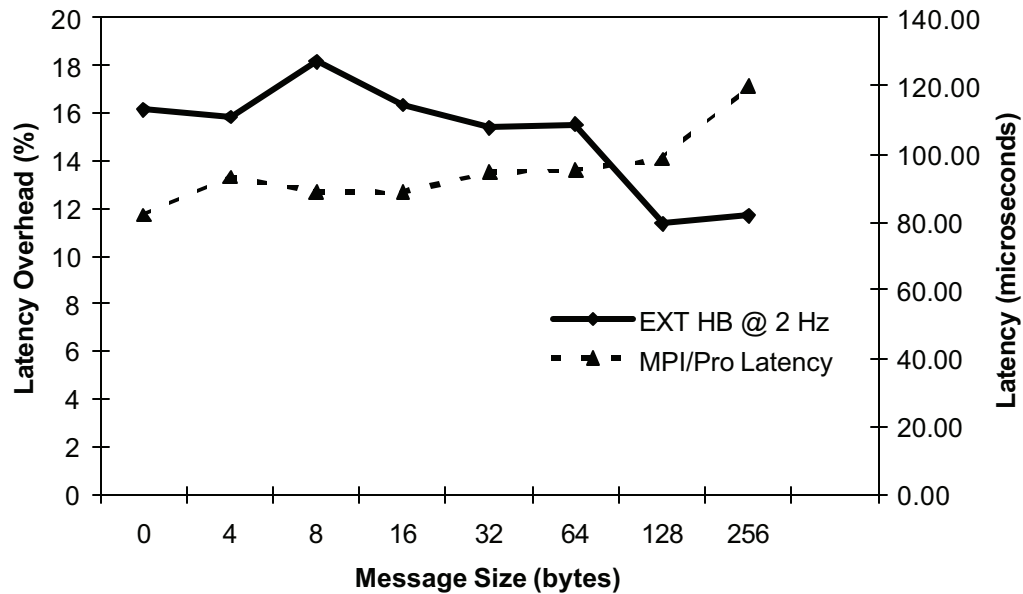


Figure 5.4: MPI/Pro Latency and Latency Overheads for smaller message sizes

5.1.2 Bandwidth Overhead

Fault-free overhead of bandwidth is defined as the percentage decrease in bandwidth in the absence of faults. The definition is given by equation 5.2,

$$Overhead_{Bandwidth} = \left(\frac{\beta_{MPI/Pro} - \beta_{MPI/FT}}{\beta_{MPI/Pro}} \right) \times 100, \quad (5.2)$$

where $\beta_{MPI/FT}$ is the bandwidth for a message size on MPI/FT and $\beta_{MPI/Pro}$ is the bandwidth on MPI/Pro middle ware for the same message size.

5.1.2.1 Experimental Setup

Figure 5.5 presents the pseudo-code for measuring bandwidth. Bandwidth is obtained by dividing the message size by the one-way latency previously measured. Results were separately obtained from a ping-pong test program compiled with MPI/Pro

and MPI/FT middleware. Fault-free overhead was then calculated as the percentage decrease in MPI/FT case when compared to MPI/Pro results. These results were computed for message sizes ranging from 0 to 1 megabyte. Results were also computed for various rates of external and internal heartbeats. The ping-pong test was run on a cluster of Intel Pentium machines (750mhz, 512 MB RAM, Linux 2.4 OS).

```

if(rank == 0){
    MPI_Send_init(sendBuff, DATASIZE, ..);
    MPI_Recv_init(recvBuff, DATASIZE, ..);
    starttime = MPI_Wtime();
    for(j = 0; j < numTests; j++){
        MPI_Start(&sendReq);
        MPI_Start(&recvReq);
        MPI_Wait(&sendReq, &status);
        MPI_Wait(&recvReq, &status);
    }
    endtime= MPI_Wtime() ;
    time = (endtime-starttime)/numTests;
    bandwidth = dataSize/(time/2)/1024/1024;//mbps
}else{
    MPI_Send_init(recvBuff, dataSize, ..);
    MPI_Recv_init(recvBuff, dataSize, ..);
    for(j = 0; j < numTests; j++){
        MPI_Start(&recvReq);
        MPI_Wait(&recvReq, &status);
        MPI_Start(&sendReq);
        MPI_Wait(&sendReq, &status);
    }
}
}

```

Figure 5.5: Pseudo-code for measuring Bandwidth

The introduction of fault-detection features and of additional checks in the path of message-passing was expected to decrease bandwidth. It was hypothesized that acceptable fault-free overhead of bandwidth would be 5%.

5.1.2.2 Results and Analysis

Bandwidth experiments were performed for various combinations of external and internal heartbeat rates. Figure 5.6 shows bandwidth overhead results for various rates of external heartbeats, while internal heartbeats were disabled. Figure 5.7 presents results for various internal heartbeat rates while the external heartbeat rate is set at 0.25 Hz.

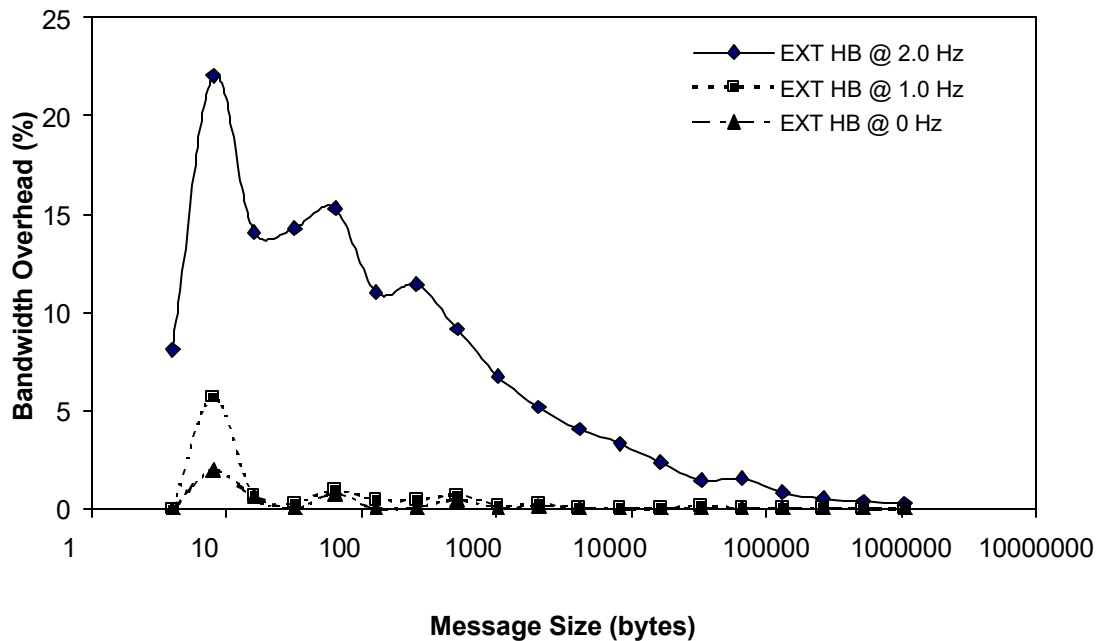


Figure 5.6: Bandwidth Overheads with only External Heartbeats

Results are similar to latency results and confirm increasing overheads with increasing rates of heartbeats. Again, the overheads decrease at longer messages, which can be attributed to the busy time concept of SCT. This similarity in results is expected as bandwidth is defined in terms of latency. Fault-free overhead of bandwidth is low at normal rates of heartbeats, and hence this supports hypothesized low overheads.

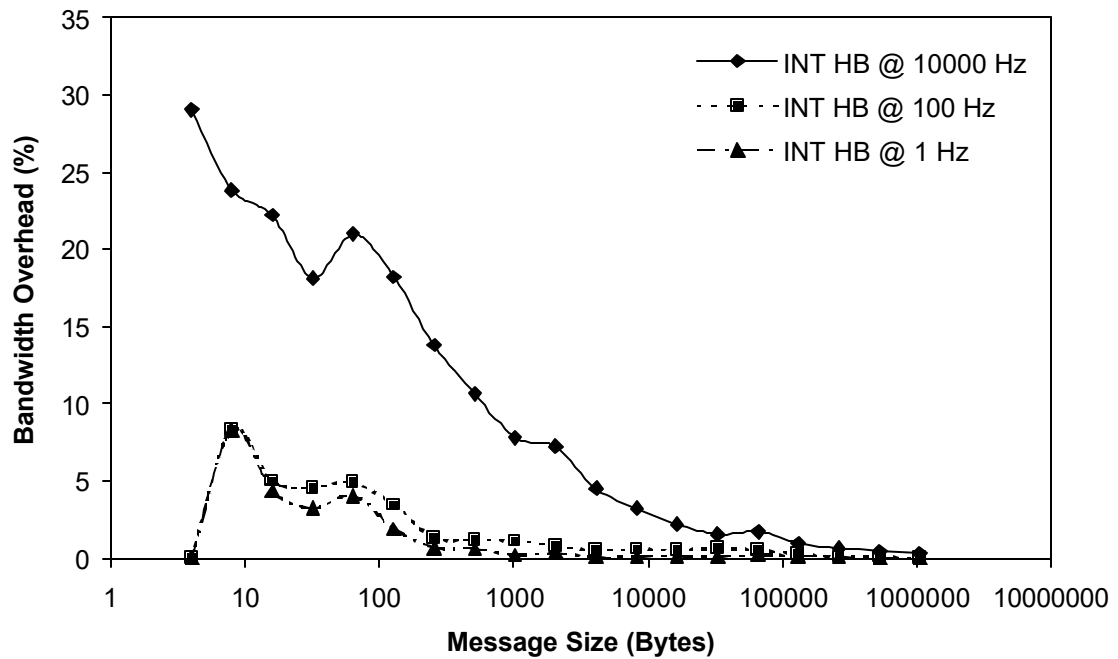


Figure 5.7: Bandwidth Overheads with Internal and External Heartbeats

Message-passing overheads for both latency and bandwidth indicate low-overheads in the absence of faults under normal rates of heartbeats. These are in line with the hypothesized values.

5.2 Model-Ia Results

Model-Ia deals with simple master/slave applications with virtual star topology. The `pmandel` program, a parallel MPI version of Mandelbrot set visualization program [23], has been used as an example for this model. Measurements for Model-Ia include both fault-free and fault-injected overheads.

5.2.1 Runtime Overhead

Fault-free overhead in runtime is defined as the percentage increase in runtime of a modified MPI parallel application on a modified middleware with respect to an

unmodified parallel application on an unmodified middleware in the absence of faults. Runtime overhead is defined in equation 5.3,

$$Overhead_{Runtime} = \left(\frac{R_{MPI/FT} - R_{MPI/Pro}}{R_{MPI/Pro}} \right) \times 100, \quad (5.3)$$

where $R_{MPI/Pro}$ is the runtime of unmodified application and $R_{MPI/FT}$ is the runtime of the modified application running on modified middleware. Fault-free overhead measurements for latency and bandwidth capture only the message-passing overheads. The following experiments are designed to capture the combined overheads of message-passing and MPI/FT API.

5.2.1.1 Experimental Setup

Simplified pseudo-code for a Model-Ia application is presented in Figure 5.8. Code in bold typeface represents the newly introduced API and other supporting functions to make the application fault-tolerant. Code marked under sections B and C performs middleware and application recovery procedures upon the death of a slave. This code section is not executed in the absence of faults. Code presented in section A represents the actual work of the master process.

Experiments measured the runtime of an unmodified program on MPI/Pro middleware. Unmodified programs are represented by the non-bold typeface code in Figure 5.8, and runtime was timed for section A. Later, runtime of modified programs (Sections: A –(B+C)) running on MPI/FT Model-Ia middleware was measured. Each program was run with a total of four processes, and the input image to be rendered was the same for both unmodified and modified applications. The rates of external and

internal heartbeats for fault detection were set at 1 Hz and 3 Hz, respectively. Applications were run on a cluster of Intel Pentium machines (900mhz, 640 MB RAM, Linux 2.4) interconnected with 10/100 Fast Ethernet.

```

if( rank == master){
    Create job array;
    Send_Init();
    Send_jobs();
    Save_jobs ();
    While (! Jobs done){
        MPIFT_GetDeadRanks (... , ... , ...)
        if (deadcount >1){
            for (I = 0.. Deadcount-1){
                Recover_job ();
                MPIFT_RecoverRank (...);
                Send_Init ();
                Send_jobs ();
                Save_jobs ();
            }
        }else{
            // Normal operation of Master
            Recv_Results ();
            Delete_Saved_jobs ();
            Send_jobs ();
            Save_jobs ();
        }
    }
    Send_End ();
    Master_Cleanup ();
}
else { // slave
    Recv_Init ();
    While( ! Recv_End ()) {
        Recv_jobs ();
        Process_jobs ();
        Send_Results ();
    }
    Slave_Cleanup ();
}

```

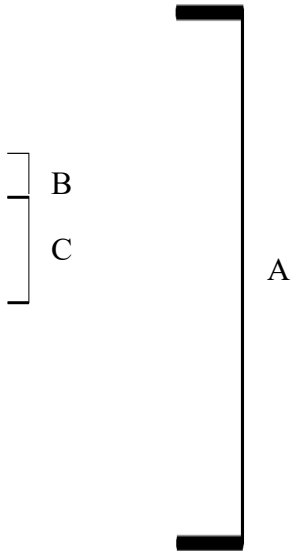


Figure 5.8: Pseudo-code for Model-Ia Application

5.2.1.2 Results and Analysis

Table 5.1 presents the results of running the program with and without MPI/FT at various process sizes. Runtime overhead for Model-Ia pmandel application was found to

be within acceptable limits in all the cases. It supports the fact that applications can have simple, effective fault-tolerance and yet have small fault-free overhead. These results support the hypothesis.

Table 5.1: Fault-free Runtime Overhead in Model-Ia, Pmandel Application

Number of Processes	Unmodified Runtime (seconds)	Modified Runtime (seconds)	Hypothesized Overhead (%)	Actual Overhead (%)
2	2.969	3.030	15	2.05
3	1.511	1.578	15	4.43
4	1.041	1.102	15	5.86

5.2.2 Program changes

MPI applications need to be modified to utilize the MPI/FT API and enable fault-tolerance. The amount of changes required for sample applications is studied in this section.

5.2.2.1 Program Change Ratio

Program change ratio (PCR) is defined as ratio between number of new FT API calls introduced and number of lines of code in original code. It is a simple measure to study the amount of changes required in original programs to utilize MPI/FT.

5.2.2.2 Results and Analysis

The values pertinent for the pmandel program are presented in Table 5.2.

Table 5.2: Program Change Ratio in Model-Ia, Pmandel Application

Original lines of code	New FT API	Hypothesized PCR (%)	Actual PCR (%)
735	2	10	0.2

PCR was found to be well below the expected values because the number of MPI/FT API calls for achieving fault-tolerance is dependent on the structure of the program and not on the original lines of code. New code accounting for application level recovery support is application dependent and does not affect PCR values.

5.2.3 Recovery Time

The experiments in this section are focused on showing that applications recover from externally introduced faults, and the recovery times are within acceptable limits. Recovery time encapsulates the time where the progress of the parallel application is stalled by fault recovery and repair procedures. Recovery time consists of middleware and parallel application components. Middleware recovery time refers to the time taken by the middleware to restart and incorporate a process into the process group. While a small middleware recovery time is essential for high performance applications, predictable recovery time will play a vital role for real-time parallel applications. Application recovery time refers to the time spent by application-specific recovery procedures. These recovery procedures are primarily focused on initializing the new spare process into the required state.

5.2.3.1 Experimental Setup

Figure 5.8 presents the pseudo-code for pmandel application. Section B marks the code for middleware recovery procedures, and section C refers to the application recovery part. Experiments were performed by running a modified pmandel application with four processes under the following conditions:

- 1) Fault-free run with MPI/Pro.

- 2) Fault-free run with MPI/FT.
- 3) Single slave failure after X% of the pixels are computed and recovery is performed.
- 4) Single slave failure after X% of the pixels are computed and recovery is not performed. Application continues with the remaining 2 slaves.

The value of X was varied from 10 to 90 in increments of 10. Faults were simulated by programming termination of slaves through messages internal to the pmandel application. The times for middleware recovery and application recovery were measured by utilizing `MPI_Wtime()`. The rates of external and internal heartbeats for fault detection were set at 1 Hz and 3 Hz, respectively. Applications were run on a cluster of Intel Pentium machines (900mhz, 640 MB RAM, Linux 2.4) interconnected with 10/100 Fast Ethernet.

5.2.3.2 Results and Analysis

The pmandel application successfully recovered from the single faults introduced. Middleware and application-level recovery were successfully performed, and the application progressed to a successful completion. This successful completion proves part of the hypothesis that applications can successfully recover and complete in presence of external faults.

The average results for middleware and application-level recovery for single slave deaths are presented in Table 5.3. The results show that actual middleware recovery time is small. Small recovery times are essential in reducing overall runtime in faulty environments. Figures 5.9 and 5.10 show the progress of the application when faults are

introduced at 10% and 90% progress of the application. Runtime results with faults introduced at various other rates of application progress are presented in Appendix A.

Table 5.3: Recovery Time in Model-Ia, Pmandel Application

Hypothesized Middleware Recovery Time (milliseconds)	Actual Middleware Recovery Time (milliseconds)	Hypothesized Application Recovery Time (milliseconds)	Actual Application Recovery Time (milliseconds)
25	24.12	500	16.34

It is evident from the graphs that performing recovery depends on the current progress of the application. While it is beneficial to recover a dead rank during the initial part of an application's progress, there is no benefit to perform recovery towards the end of an application's progress. In fact, there is a penalty. For this particular experiment such a penalty for recovering a dead worker is evident in cases when the death of the process occurs after 60% of application progress.

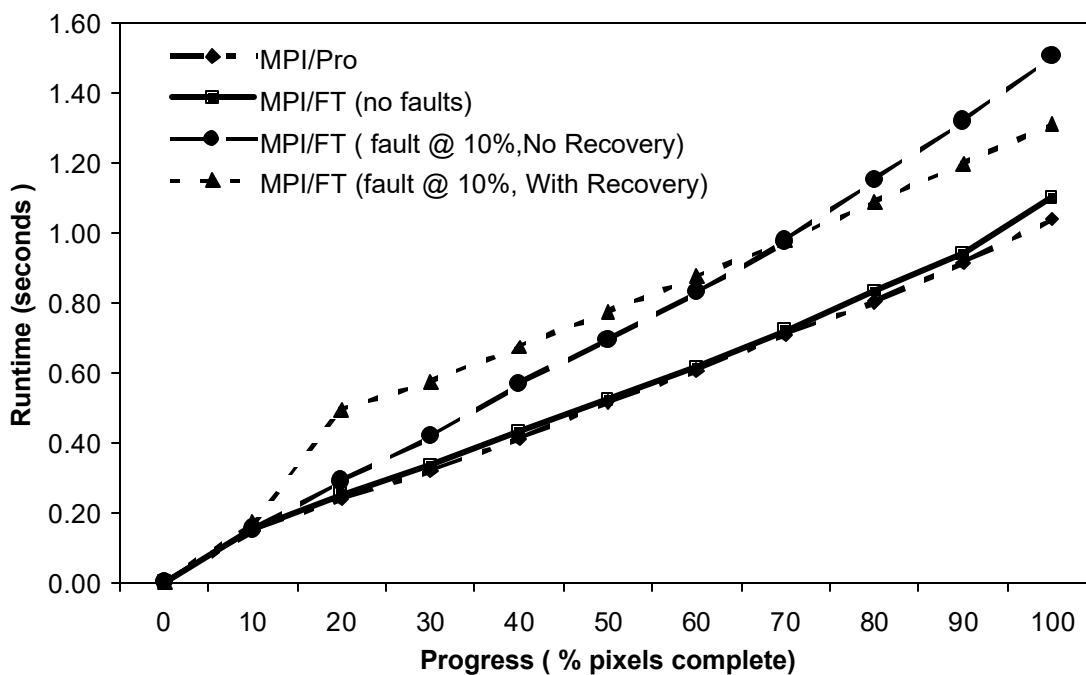


Figure 5.9: Runtime of Pmandel Application with faults at 10 % of Progress

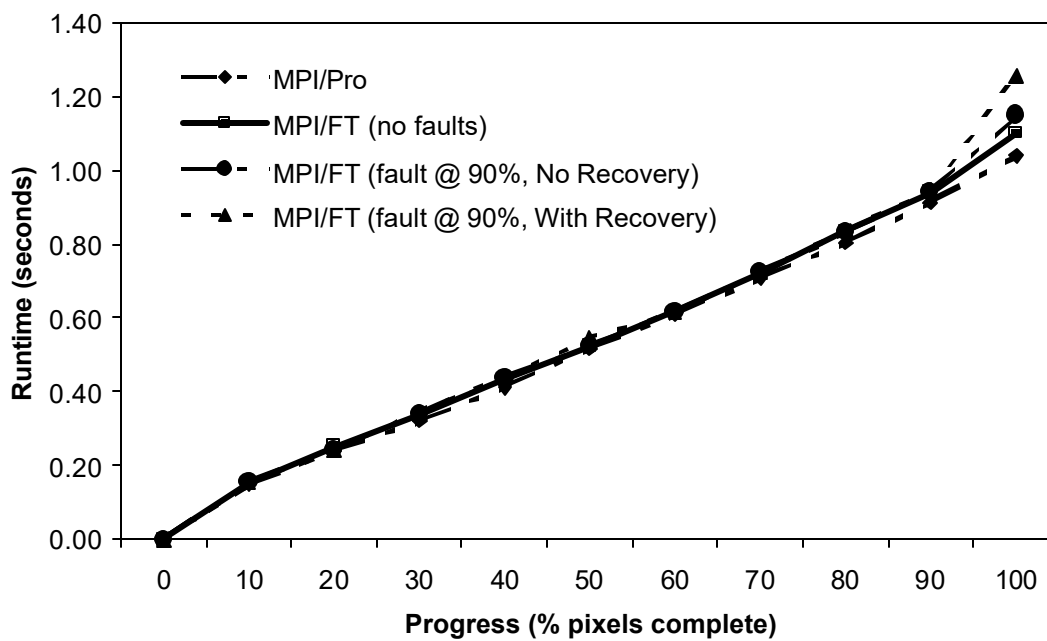


Figure 5.10: Runtime of Pmandel Application with faults at 90 % of Progress

In summary, Model-Ia results aid in proving part of the hypothesis for one of the two models specified. Runtime and PCR results prove that applications can have low fault-free overheads. Recovery time experiments show that applications recover from external faults to successfully complete. Middleware and application recovery were also performed within acceptable time limits.

5.3 Model-IIa Results

Model-IIa consists of applications with SPMD all-interacting style and virtual all-to-all topology. The example program used for this model is the Game of Life [6], a discrete event simulation program that requires communication with each of its neighbor processes. Additionally, Model-IIa applications are expected to run in an iterative fashion. Similar to Model-Ia, Model-IIa measurements include both fault-free and fault-injected overheads.

Model-IIa applications also utilize user-aware checkpointing to save the application-relevant state at predetermined synchronization points. Checkpointing in these applications is user-initiated, and state of application is determined by the data contents of the application. Model-IIa applications typically run in an iterative fashion, and checkpointing may be performed at the end of an iteration. Checkpointing operations are costly and incur high overhead as they typically involve disk accesses and synchronization of all processes in applications. As expected, the overhead incurred for a single checkpoint is several orders of magnitude of the time taken for a single iteration. In such a case checkpointing must be judiciously used.

Checkpointing frequency is defined as the ratio between the number of checkpoints performed and the total number of iterations in the application. Invoking checkpointing

routines at a greater frequency than required leads to unnecessary fault-free overhead. In the presence of faults, a lower-than-required frequency results in the lack of "fall through," or real progress of the application. Appropriate time between checkpoints depends on several factors such as distribution of faults, time to checkpoint, and time to recovery. This issue has been dealt with in literature [21, 28, 32]. Model-IIa results will thus be presented with varying checkpointing frequencies where appropriate.

5.3.1 Runtime Overhead

The definition of runtime overhead is the same as defined in Model-Ia results and captures percentage increase in run-time. However with regard to the preceding discussion on checkpointing frequency, runtime overheads will be measured for a range of checkpointing frequencies. Percentage increase in runtime is computed with respect to runtime of an unmodified program running on plain MPI/Pro middleware.

5.3.1.1 Experimental Setup

Figure 5.11 presents pseudo-code for a Model-IIa application. The code in bold typeface consists of the newly introduced API and supporting code for making the application fault-tolerant. Sections B and C perform middleware and application level recovery in the event of a process death. Section D presents code for marshalling application state and checkpointing the information. Checkpointing frequency of the application is passed through command line arguments. Section A captures the code with the actual work and other fault-tolerant relevant code.

Modified and unmodified applications consisted of four processes running on a logical 2x2 grid topology and 10,000 iterations of the Game of Life. Runtime of an

unmodified application (Section A – code in bold typeface) running on unmodified middleware was measured. Later, runtimes of modified applications (Sections: A – (B+C)) were determined. The variants in later applications consisted of checkpointing frequency and data grid sizes. Checkpointing frequency, as defined in Section 5.3, was varied from 0 through 0.01, and data grid sizes were varied from 4x4 to 250x250 elements for each application run. Timing measurements were made by utilizing the `MPI_Wtime()` function. Applications were run on a cluster of Intel Pentium machines (750mhz, 512 MB RAM, Linux 2.4 OS) interconnected with 10/100 Fast Ethernet.

```

Distribute Data;
Initialize conditions;
While (! enuf_iterations){
  MPIFT_GetDeadRanks(..);
  if (deadcount >1){
    for (I = 0.. Deadcount-1){
      MPIFT_RecoverRank(..);
    }
    MPIFT_ChkptRecover(&retrieved, ..);
    Restore_AppState(&retrieved);
  };
  //normal run part
  Communicate_part();
  Compute_part();
  MPI_Barrier();
  if ( func(chkpt_freq) ) {
    Get_AppState(&tostore);
    MPIFT_ChkptDo(&tostore, .., ..);
  }
};
Cleanup();

```

Figure 5.11: Pseudo-code for Model-IIa Application

5.3.1.2 Results and Analysis

Table 5.4 presents the overhead results without checkpointing. These overheads measure the absolute performance impact of message-passing overheads and introduction

of new MPI/FT API on the application runtime. These results indicate that overheads are low and decrease for larger grid sizes with more computation. This decrease may be attributed to the different impacts of FT mechanisms on communication and computation and are further explored in Chapter 6.

Figure 5.12 presents runtime overheads with various checkpointing frequencies (0 to 0.01). The number of actual checkpoints is equivalent to product of the frequency and the number of total iterations. It can be realized from the graph that overheads depend on the size of data to checkpoint and their frequency. Overheads are limited under moderate checkpoint frequency.

Table 5.4: Fault-free Runtime Overhead without Checkpointing in Model-IIa, Game of Life Application

Grid Size	Runtime with MPI/Pro (seconds)	Runtime with MPI/FT (seconds)	Hypothesized Overhead (%)	Actual Overhead (%)
2x2	8.84	9.26	30	4.7
16x16	8.97	9.43	30	5.2
100x100	10.63	11.17	30	5.0
250x250	20.75	21.25	30	2.4

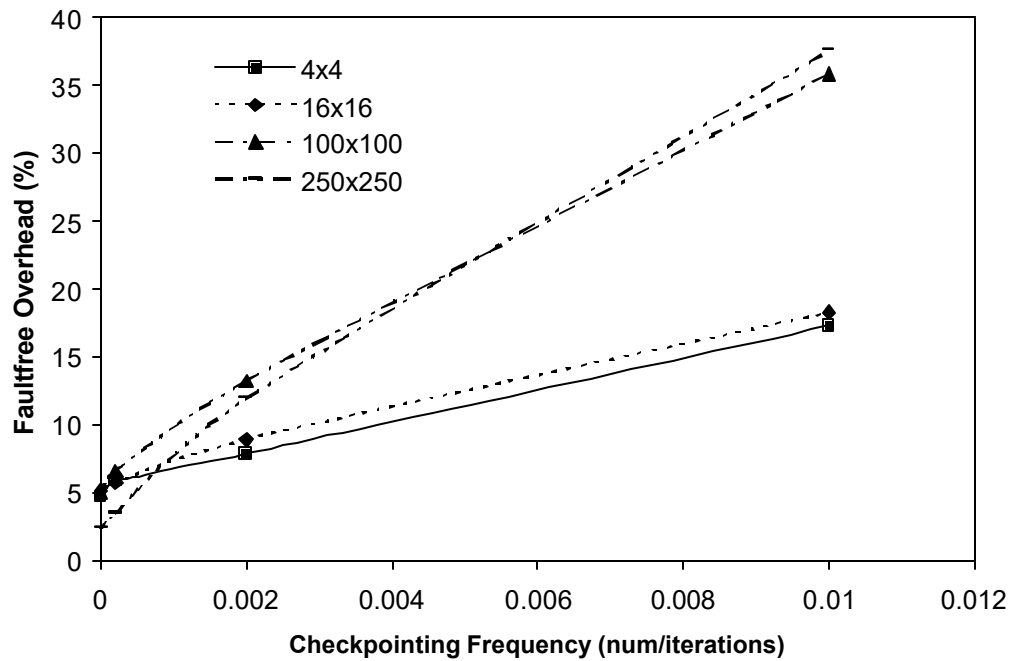


Figure 5.12: Fault-free Overhead with Checkpointing in Model-Ia, Game of Life Application

5.3.2 Program Changes

MPI applications need to be modified to utilize the MPI/FT API and enable fault-tolerance. The amount of changes required for sample applications is studied in this section.

5.3.2.1 Program Change Ratio

The definition of PCR is the same as for Model-Ia.

5.3.2.2 Results and Analysis

The values pertinent for the Game of Life program are presented in Table 5.5. PCR was found to be well below the expected values because the number of MPI/FT API calls

for achieving middleware level recovery was dependent on the structure of the program and not on the original lines of code. New code that accounts for application level recovery is application dependent and does not affect PCR.

Table 5.5: Program Change Ratio for Model-IIa, Game of Life Application

Original lines of code	New FT API	Hypothesized PCR (%)	Actual PCR (%)
538	4	10	0.75

5.3.3 Recovery Time

As explained in Model-Ia results, recovery time consists of both middleware and application-level recovery components. Experiments were designed to measure each of these components individually.

5.3.3.1 Experimental Setup

Figure 5.11 presents the pseudo-code for Model-IIa application. Section B emphasizes the middleware recovery procedure, and section C refers to the application recovery part. Experiments were performed by running a modified Game of Life application with four processes on a 2x2 process grid. External faults were simulated by manual termination of non-rank 0 processes. The times for middleware and application recovery were measured by utilizing `MPI_Wtime()` function. The rates of external and internal heartbeats for fault detection were set at 1 Hz and 3 Hz, respectively. Applications were run on a cluster of Intel Pentium machines (750mhz, 512 MB RAM, Linux 2.4 OS) interconnected with 10/100 Fast Ethernet. Middleware recovery time was

expected to be around 50 milliseconds, and the applications recovery time was expected around 0.5 seconds.

5.3.3.2 Results and Analysis

The Game of Life [6] application successfully recovered from externally introduced single faults. Middleware and application recovery were successfully performed and enabled the application to continue to a successful completion. Table 5.6 presents the average values for middleware recovery of a single dead process. Though the actual value exceeds the hypothesized value, the actual recovery time is still small. This increase in recovery time can be attributed to the collective recovery of a dead rank in Model-IIa.

Table 5.6: Middleware Level Recovery Time for Model-IIa, Game of Life Application

Hypothesized Middleware Recovery Time (milliseconds)	Actual Middleware Recovery Time (milliseconds)
50	72

Table 5.7: Application Level Recovery Time for Model-IIa, Game of Life Application

Grid Size	Hypothesized Application Recovery Time (milliseconds)	Actual Application Recovery Time (milliseconds)
4x4	500	2.3
16x16	500	3.0
100x100	500	4.2
250x250	500	4.8

Table 5.7 presents the application level recovery time for the Game of Life application after a process is recovered at middleware level. This recovery time includes both retrieval of checkpointed information and restoring application state. It can be seen that the time required to recover depends on the size of the data to be read. These values are less than the hypothesized value of 500 milliseconds for application recovery.

5.4 Summary

Experiments from both fault-injected and fault-free categories yielded acceptable values to prove the hypothesis. Fault-injected experiments proved that applications recover from externally introduced faults and successfully run to completion. Low middleware and application recovery times for both Model-Ia and Model-IIa applications indicate the quick recovery of common applications. These values also indicate the usability of the middleware in production settings.

Fault-free measurements proved that overheads and modifications in the absence of faults are low and within acceptable values. The common message-passing overhead results for both Model-Ia and Model-IIa indicate the low latency and bandwidth overheads at normal rates of internal and external heartbeats. The message-passing overheads also indicate the increasing overheads with increasing rates of heartbeats. It must be noted that these overheads are based on a predetermined fault model, and such low overheads cannot be guaranteed for all fault-models, especially those dealing with faults in the communication domain.

Runtime overheads are low for both Model-Ia and Model-IIa sample applications. Model-IIa experiments stress the importance of the checkpointing frequency. These results show that runtime overheads are low under moderate rates of checkpointing

frequency. Low PCR values for sample applications of Model-Ia and Model-IIa indicate the minimal changes required in programs to achieve fault-tolerance. PCR values were determined to be dependent on the structure of the code rather than the original number of lines of code. Low and acceptable overhead values in message-passing, runtime, and PCR collectively prove that MPI programs can be made fault-tolerant with low fault-free overheads.

Thus, both the fault-free and fault-injected parameters and experiments validate the hypothesis.

Chapter VI

OTHER OBSERVATIONS

Previous chapters have presented parameters that assess the impact of achieving fault-tolerance on performance. Experiments were performed, and the results were used to validate the hypothesis. This chapter aims at describing observations and experiments that provide insight into performance factors and middleware design issues. Subsequent sections discuss components of fault-free overhead and some design issues through placement of services.

6.1 Components of Fault-free Overhead

MPI applications consist of communication and computation components. FT mechanisms, mainly detection mechanisms, impact communication and computation components differently. For example, detection and masking measures to remove faults in the communication channel are expected to impact communication more than computation, while masking measures to avoid memory bit flips can be expected to affect the computation component.

The CC-Ratio is a simple measure to highlight different impact of fault-detection mechanisms on communication and computation components. The CC-Ratio is derived primarily from the (C/C) Ratio [11]. (C/C) Ratio for a given execution of an application is defined as the ratio between communication cost and computation cost.

A small (C/C) Ratio is essential for scalable applications [11] and is a useful concept to predict runtime, and it aids in design of applications. Although (C/C) Ratio is a useful concept, it has limitations to separate the communication and computation components and individually understand their impact. In a typical message-passing middleware, communication costs (latency and bandwidth) are different for different message sizes, and the total communication costs may have involved messages of several sizes. The CC-Ratio presents a modified definition to capture this information and is defined as the ratio between the number of messages sent and the total number of computations performed. The message size and measure of a single computation are fixed for a run.

A sample application "Simul" is introduced to measure the CC-ratio. Simul is a master-worker MPI application, where the worker process simulates workload upon receipt of a message from the master. While Simul does not utilize MPI/FT API and is not designed to be fault-tolerant, its main goal is to highlight different impacts of FT mechanisms on communication and computation components. Both the number of messages to each slave (communication component) and the workload upon receipt of each message (computation component) are configurable for each run. This program defines a single computation workload as consisting of finding the square root and then square of ten floats and a single communication load as consisting of sending an MPI message of 40 bytes as payload. Thus, if a Simul program sends 100 messages to a slave, and upon receipt of each message the slave performs 10 computations, the CC-Ratio for this run is set at 100/10. It must be noted that the definitions of a single computation and communication load have been arbitrarily set, and they can be refined to a standard definition.

Experiments were performed with a Simul program with a master and a single worker process. Communication and computation components were varied from 1 to 1,000,000, with their product remaining a constant at 1,000,000. These experiments were run with three different middlewares. Runtime results were obtained by running the Simul program on the baseline MPI/Pro middleware. Similar runtimes were obtained by running the Simul program on an MPI/FT realization for Model-Ia. Later, results were obtained by running the Simul program on a debug-enabled variant of MPI/FT. This middleware is expected to introduce overheads in computation because of the debugging routines and assertions.

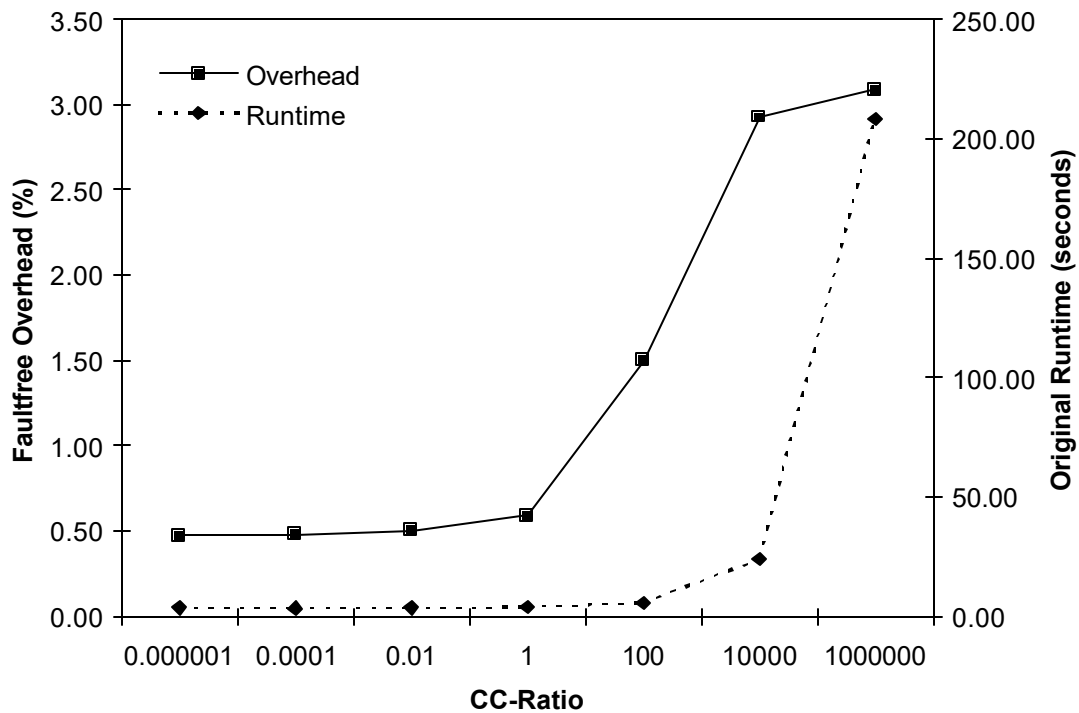


Figure 6.1: Fault-free Overhead and CC-Ratio for MPI/FT Model-Ia at 40 bytes

Figures 6.1 and 6.2 present the fault-free overhead of the second and third middlewares as compared with the baseline MPI/Pro middleware. In all the runs, the external heartbeats were set 1Hz, while internal heartbeats were disabled. Simul was run on a cluster of Intel Pentium machines (750mhz, 512 MB RAM, Linux 2.4 OS). Results suggest that MPI/FT Model-Ia middleware impacts communication more than computation. Programs with a higher CC-Ratio have higher overheads than with lower values of the CC-Ratio. Conversely, the debug-enabled variant of MPI/FT Model-Ia middleware impacts computation more than communication.

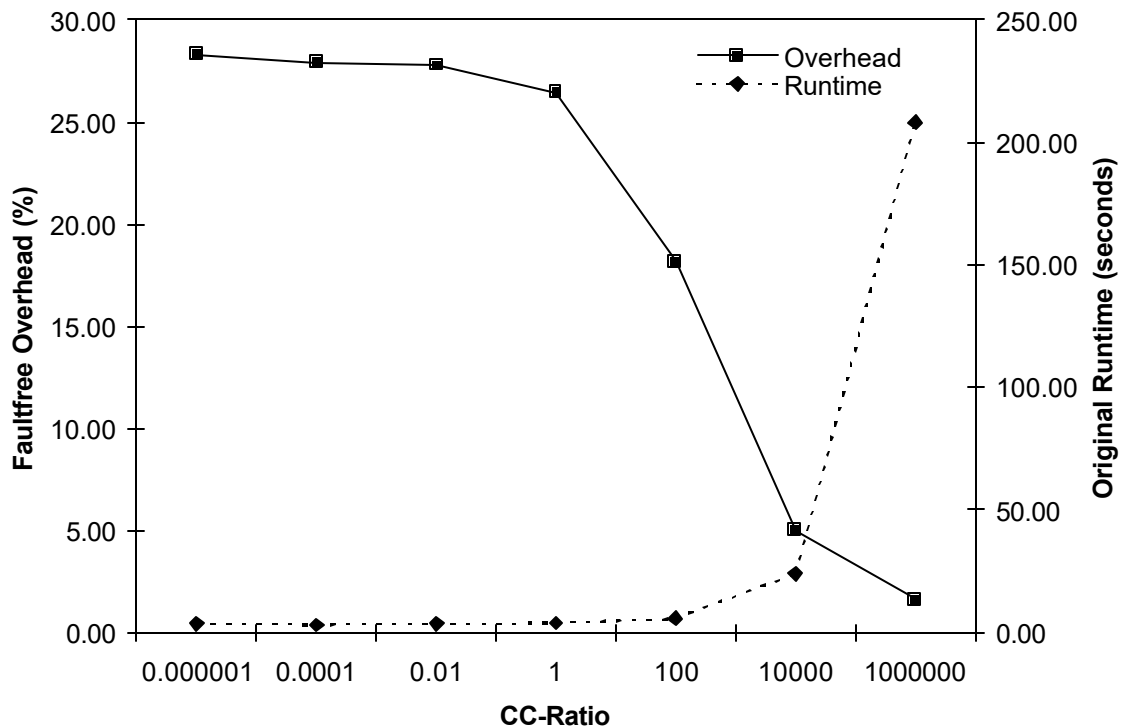


Figure 6.2: Fault-free Overhead and CC-Ratio for Debug-enabled MPI/FT Model-Ia
at 40 bytes

Apart from these observations, the concept of the CC-Ratio can be useful in several ways:

- It leads to the realization that describing fault-tolerant overheads of a middleware with a single fixed application is insufficient.
- Experiments with varying message sizes and standard definitions of computation workload can determine the overhead profile of a middleware for a given system.
- The profile of a middleware can guide users with predetermined fault-models to design a program with an appropriate CC-Ratio. Conversely, for a program with fixed CC-Ratio the profiles of several middlewares can be compared to choose an appropriate middleware.

6.2 Placement of Services

Fault-tolerance consists of the primary steps of detection, notification, and recovery. This subsection discusses the placement of these steps across various levels of the system. Mechanisms for detection and recovery are primarily determined by the fault model and fault rate. Detection of faults can be performed across various components in the system. Each of these levels is adept at performing detection for certain faults. Faults at one layer typically manifest as different faults at another layer. As expected, detection of faults can take place at various levels, but the cost associated with such detection depends on the kinds of faults and their rate. Costs for detection can be incurred either as run-time (temporal) or resources (spatial: memory, hardware).

As an example, consider a system that has to detect externally introduced memory bit flips. Such faults are common in space-based missions. These memory bit flips can be either tolerated or masked at the hardware level by using special hardware (parity/ECC memory). Usage of such specialized hardware might be prohibitive in cases of COTS components. In such a case, the OS level can potentially check for such bit flips. The OS can mask the faults by replicating the process state or by periodically performing checksum on parts of the process. However, these approaches may incur high overheads. In the absence of an OS level detection or masking, middleware will need to perform similar operations. The application level is the appropriate place in several cases for such memory flip detection. The assumption is that these memory flips have reached the application layer and have not affected the process. Methods such as ABFT [18, 31] can be effectively used to check the validity of data. Since applications are aware of consistency of data, only memory flips that affect the data in a perceptible manner will be detected.

Similarly, recovery can be achieved across various levels and can be designed across several layers as in detection. Recovery achieved at lower layers and transparent to the user typically requires either replication and/or system-level checkpointing. Replication of message-passing programs is non-trivial. This research effort advocates the usage of recovery mechanisms that are achieved by an interaction of middleware and application level. Such a strategy allows applications to determine their own recovery policy. For message-passing middleware, an application can choose to recover a process immediately, delay the recovery, or request the middleware to shrink the world size. Such

policies may be based on the current fault-rate, application stall time for recovery, and overall application progress.

A good design for a fault-tolerant system must thus consider placement of services across various levels. Each of these components and their interaction can have different costs and effectiveness in achieving the required feature. An appropriate mix and match of features across these layers will be essential in achieving a low overhead fault-tolerant system.

Chapter VII

CONCLUSIONS

7.1 Summary

This thesis hypothesized that modified MPI applications running on a modified middleware will complete successfully even in the presence of faults. It also hypothesized that these modified applications will incur acceptable performance in the absence of faults. The motivation, hypothesis, and contribution expected from this thesis were presented in the first chapter. MPI and Cluster-based systems have proliferated into various domains of usage. Their lack of reliability has hampered their usage in critical environments. High costs of failure in these systems and the lack of existing efforts motivated the need for a fault-tolerant and reliable MPI implementation. The hypothesis was supported through the design and implementation of MPI/FT. MPI/FT [4] was realized by selective incorporation of fault-tolerant features into MPI/Pro [26], an existing high performance implementation of MPI 1.1 standard.

The literature survey in the second chapter revealed the shortcomings of MPI 1.1 standard in addressing reliability issues. Other research efforts to make MPI more reliable were briefly described. The third chapter presented the basis for this thesis. The model-based approach is based on the realization that applications contain features and characteristics that are amenable to achieving fault-tolerance. This thesis exploited those

features to provide application model-specific services and achieve low-overhead fault-tolerance. Relevant AEMs based on parameters of model-based approach were introduced.

The fourth chapter presented the research methodology and design of MPI/FT. The design of fault-tolerant features for detection, notification and recovery was elaborated. This chapter has also presented a brief description of the MPI/FT API available to users and provided examples of their usage through pseudo code for some example applications.

The fifth chapter presented the necessary experiments to validate the hypothesis. Parameters to assess the impact of fault-tolerance on performance were defined. Design of experiments and results of fault-free overhead on messaging were presented. These experiments evaluated the impact of fault-tolerance measures on the latency and bandwidth of the message-passing at various message sizes. Experiments for two AEMs, Model-Ia and Model-IIa, were presented. These experiments were designed to evaluate the impact on the runtime of these applications. In Model-IIa applications the concept of checkpointing frequency has been introduced. Fault-free parameters in message-passing and runtime were defined and obtained to validate that low-overhead fault-tolerance was possible. Fault-injected experiments proved that applications finished successfully with limited overheads in the presence of external faults. Chapter 6 presented other observations and experiments to provide insight into performance impact of fault-tolerance mechanisms on communication and computation components.

7.2 Future work

This work has implemented two AEMs: Model-Ia and Model-IIa. Both these models have the concept of a SPA (Safe Protection Area), in which the master process/rank 0 is not affected by external faults. The initial design of MPI/FT was targeted for space-borne environments that support hardened environments and can provide such a SPA. However, the assumption of such a SPA might not be feasible in certain space-based environments and most ground applications. Future work may implement models that do not require such a SPA while incurring additional overhead.

The need for a SPA in Model-I applications can be eliminated with parallel NMR (N modular redundancy). Parallel NMR involves providing N way redundancy for a single process and can be achieved either through active or passive redundancy. Figure 7.1 shows one way of achieving parallel NMR for Model-Ic. Messages from various slaves will need to be replicated across various copies of master process.

Model-II applications are marked by a similarity in their code, and the roles of different processes are interchangeable. In Model-II, the coordinator can be placed at any rank. A simple strategy is that the coordinator be placed at rank 0, and another rank named secondary coordinator can check for the health of rank 0. Death of a rank other than rank 0 can be dealt with as in the current Model-IIa. However, death of rank 0 can be handled by the secondary coordinator. Such a revolving or secondary coordinator approach may introduce new API to determine the "recovery head" process. This recovery head process can execute the code that drives the recovery while rest of the alive ranks participate as required.

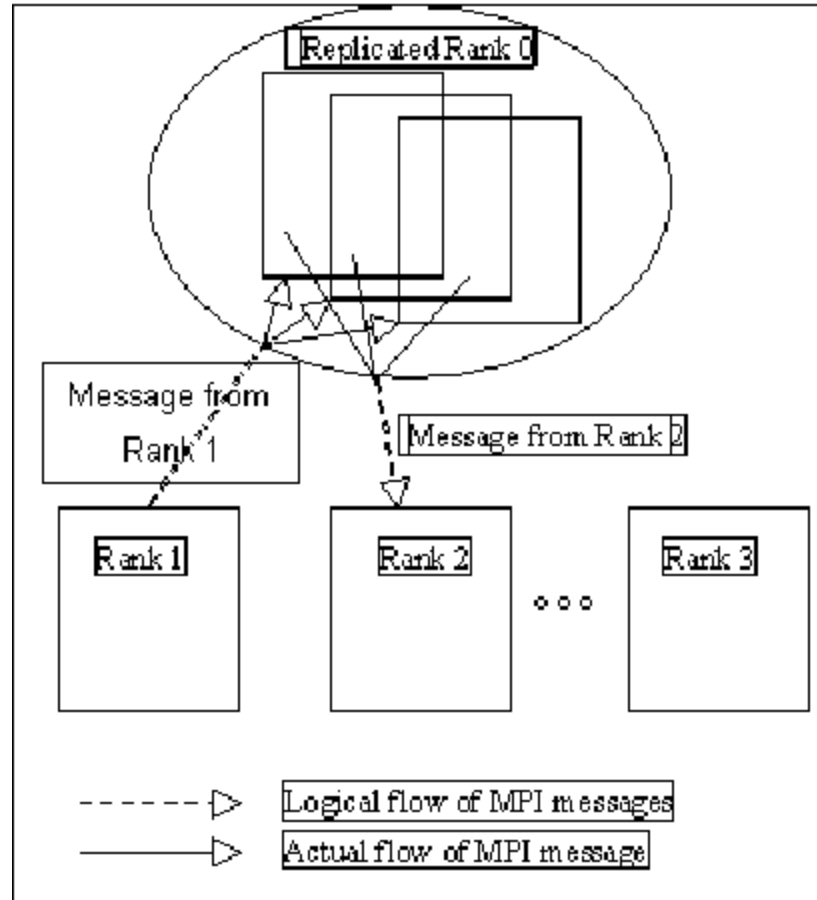


Figure 7.1: Model-Ic with Parallel NMR

Scalable fault-tolerance requires both scalable detection and recovery procedures. The current star topology at FT level for heartbeats is inherently not scalable. This star topology can be replaced by hierarchical methods with overlapping zones. Other scalable fault-detection methods based on gossiping [10] can also be explored.

Newer models that are scalable in recovery are possible. In certain applications, processes interact infrequently at a global level and more frequently among a cohort of neighboring processes. In such cases the impact of death of a process only affects processes in the cohort, and recovery procedures may also be limited to this cohort of processes.

REFERENCES

- [1] A. Agbaria, and R. Friedman, "Starfish: Fault-Tolerant Dynamic MPI programs on Clusters of Workstations," *Proceedings: 8th IEEE International Symposium on High Performance Distributed Computing*, Redondo Beach, California, Aug. 1999, pp. 167-176.
- [2] D. Bakken, "Middleware", <http://www.eecs.wsu.edu/~bakken/middleware.pdf> (current Sep. 2002).
- [3] S. Balay et al., *PETSc Users Manual*, technical report ANL-95/11, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois, 1995.
- [4] R. Batchu et al., "MPI/FTTM: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing," *Proceedings: 1st IEEE International Symposium of Cluster Computing and the Grid*, Melbourne, Australia, May 2001, pp. 26-33.
- [5] A. Beguelin, E. Seligman, and P. Stephan, *Application Level Fault Tolerance in Heterogeneous Networks of Workstations*, technical report CMU-CS-96-157, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1996.
- [6] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for your mathematical plays, Volume 1: Games in General*, Academic Press, New York, New York, 1982.
- [7] L. Birov et al., "PMLP Home Page," *PMLP Web page*, <http://hpcl.cs.msstate.edu/pmlp> (current August 2002)
- [8] G. Burns, R. Daoud, and J. Vaigl, "LAM: An Open Cluster Environment for MPI," *Proceedings: 8th Supercomputing Symposium*, Toronto, Canada, 1994, pp. 379-386.
- [9] R. Buyya ed., *High Performance Cluster Computing: Architectures and Systems, Volume 1*, Prentice Hall, Upper Saddle River, New Jersey, USA, 1999.

- [10] D.E. Collins, A. George, and R. Quander, "Achieving Scalable Cluster System Analysis and Management with a Gossip-based Network Service," *Proceedings: 26th Annual IEEE Conference on Local Computer Networks*, Tampa, Florida, pp. 49-58.
- [11] M. Crovella et al., "Using communication-to-computation ratio in parallel program design and performance prediction," *Proceedings: 4th IEEE Symposium on Parallel and Distributed Processing*, Arlington, Texas, Dec.1992, pp. 238-245.
- [12] ETNUS Inc., "TotalView User's Manual," *TotalView Specification*, <http://www.etnus.com> (current August 2001)
- [13] G. Fagg, and J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting Dynamic Applications in a Dynamic World," *Proceedings: 7th European PVM/MPI Users' Group Meeting*, Balatonfüred, Hungary, Aug. 2000, pp. 346-353.
- [14] W. Gropp et al., "MPICH: A High Performance, Portable Implementation of the MPI Message Passing Interface Standard", *Parallel Computing*, vol. 22, no. 6, Jan. 1996, pp. 789-828.
- [15] F. A. Haight, *Handbook of the Poisson Distribution*, John Wiley and Sons Inc., New York, New York, 1967.
- [16] M. Hayden, *The Ensemble System*, technical report TR98-1662, Computer Science Department, Cornell University, Ithaca, New York, 1998.
- [17] W.L. Heimerdinger, and C.B. Weinstock, *A Conceptual Framework for System Fault Tolerance*, technical report CMU/SEI-92-TR-33, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1996.
- [18] K.H. Huang, J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. 33, December 1984, pp. 518-528.
- [19] D.L. Katz et al., "Applications Development for a Parallel COTS Spaceborne Computer," *Proceedings: 3rd Annual Workshop on High-Performance Embedded Computing*, Sep. 1999, Lexington, Massachusetts.
- [20] J. F. C. Kingman, *Poisson Processes*, Oxford University Press, New York, New York, 1993.
- [21] Y. Ling, J. Mi, and X. Lin, "A Variational Calculus Approach to Optimal Checkpoint Placement," *IEEE Computers*, vol. 50, no. 7, July 2001, pp. 699 -708.

- [22] M. Litzkow et al., *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, technical report 1346, Department of Computer Science, University of Wisconsin-Madison, Madison, Wisconsin, 1997.
- [23] B.B. Mandelbrot, *The Fractal Geometry of Nature*, 2nd edition, W.H. Freeman and Co., San Francisco, California, 1982.
- [24] Message Passing Interface (MPI) Forum, "Message passing interface standard 1.1," *MPI Forum*, <http://www.mpi-forum.org/docs/mpi-1.1.ps.Z> (current Sep. 2001)
- [25] Message Passing Interface (MPI) Forum, "Message passing interface standard 2.0," *MPI Forum*, <http://www.mpi-forum.org/docs/mpi-2.0.ps.Z> (current Sep. 2001)
- [26] MPI Software Technology Inc., "MPI/Pro® for Linux," *Product pages*, http://www.mpi-softtech.com/products/mpi_pro_linux/default.asp (current Sep. 2001)
- [27] MPI Software Technology Inc., "VSI/Pro® products page," *Product pages*, <http://www.mpi-softtech.com/products/vsi-pro/default.asp> (current Aug. 2002)
- [28] J. S. Plank and W. R. Elwasif, "Experimental assessment of workstation failures and their impact on checkpointing systems," *Proceedings: 28th International Fault-Tolerant Computing Symposium*, Munich, Germany, June 1998, pp. 48-57.
- [29] S. Rao, L. Alvisi, and H.M. Vin. "Egida: An Extensible Toolkit for Low-overhead Fault-tolerance," *Proceedings: 29th International Fault-Tolerant Computing Symposium*, Madison, Wisconsin, June 1999, pp. 48-55.
- [30] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," *Proceedings: 10th International Parallel Processing Symposium*, Honolulu, Hawaii, Apr. 1996, pp. 526-531.
- [31] S.J. Wang and N.K. Jha, "Algorithm-based fault tolerance for FFT networks," *IEEE Transactions on Computers*, vol. 43, no. 7, July 1994, pp. 849-854.
- [32] K. F. Wong and M. Franklin, "Checkpointing in distributed systems," *Journal of Parallel and Distributed Systems*, vol. 35, no. 1, :67-75, May 1996, pp. 67-75.

Appendix A
OTHER MODEL-IA RESULTS

This appendix presents runtime results for Model-Ia pmandel application [23]. Experiments were performed by running a modified pmandel application with four processes under the following conditions:

- 1) Fault-free run with MPI/Pro.
- 2) Fault-free run with MPI/FT.
- 3) Single slave failure after X% of the pixels are computed and recovery is performed.
- 4) Single slave failure after X% of the pixels are computed and recovery is not performed. Application continues with the remaining two slaves.

Faults were simulated by programming termination of slaves through messages internal to the pmandel application. The times for middleware recovery and application recovery were measured by utilizing `MPI_Wtime()`. The rates of external and internal heartbeats for fault detection were set at 1 Hz and 3 Hz, respectively. Applications were run on a cluster of Intel Pentium machines (900mhz, 640 MB RAM, Linux 2.4) interconnected with 10/100 Fast Ethernet. Figures A.1 through A.7 present results when X was varied from 20% to 80% in increments of 10%.

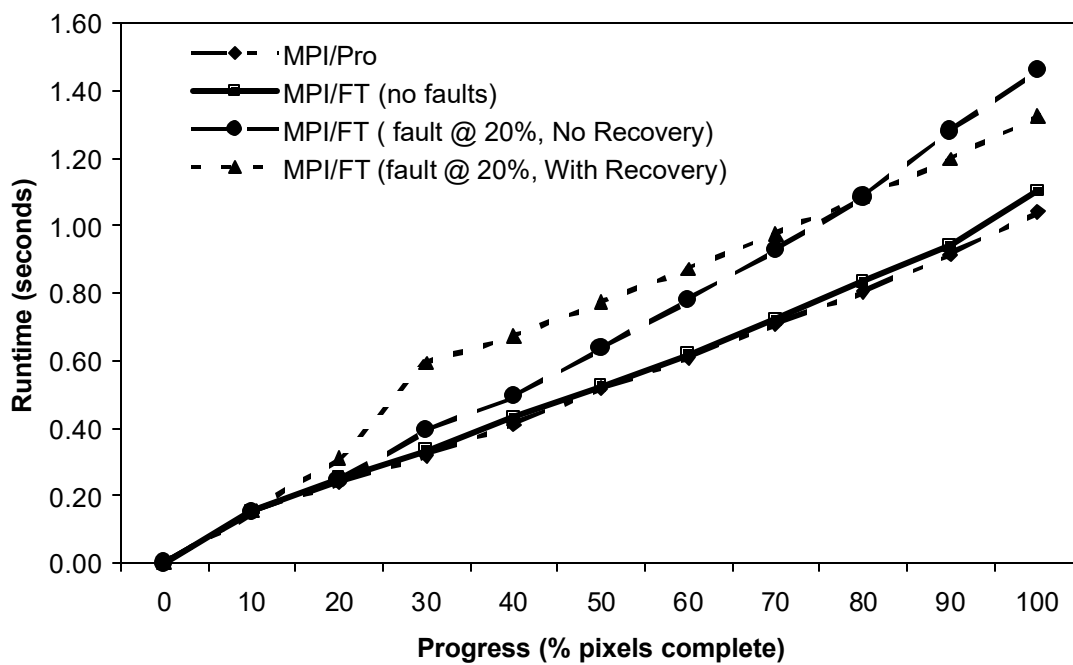


Figure A.1: Pmandel Application Runtime with Fault at 20 % Progress

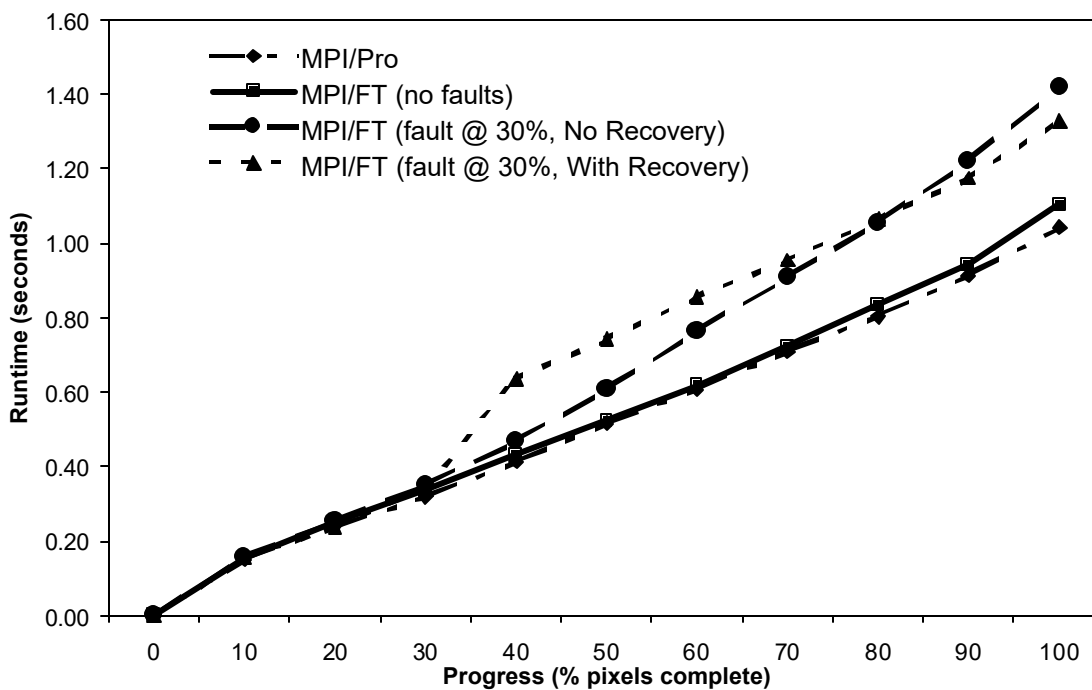


Figure A.2: Pmandel Application Runtime with Fault at 30 % Progress

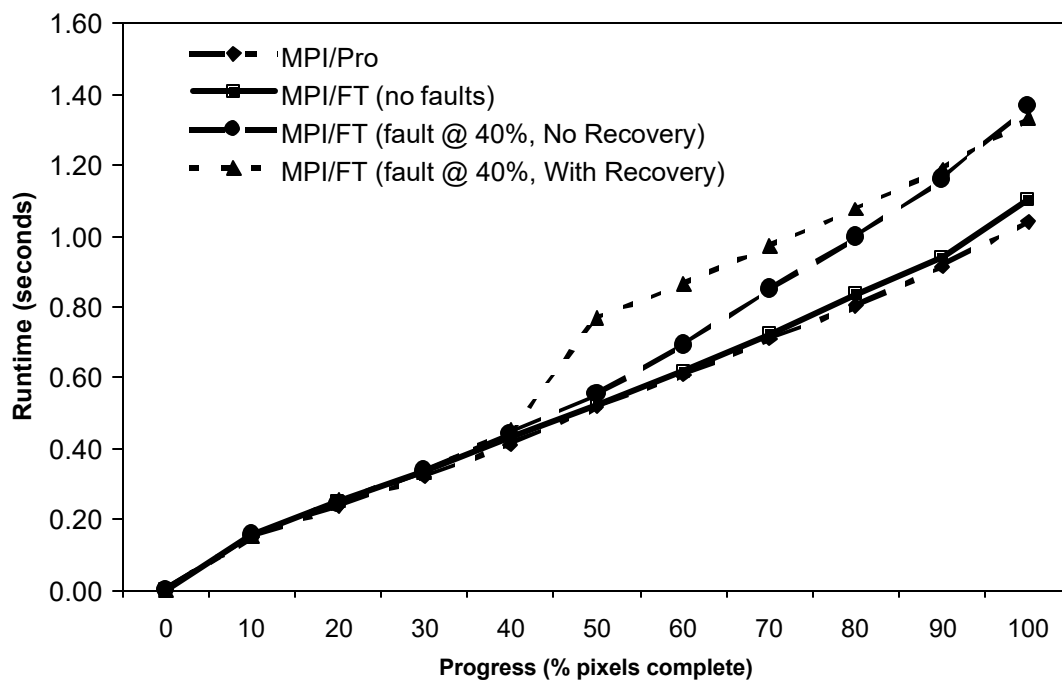


Figure A.3: Pmandel Application Runtime with Fault at 40 % Progress

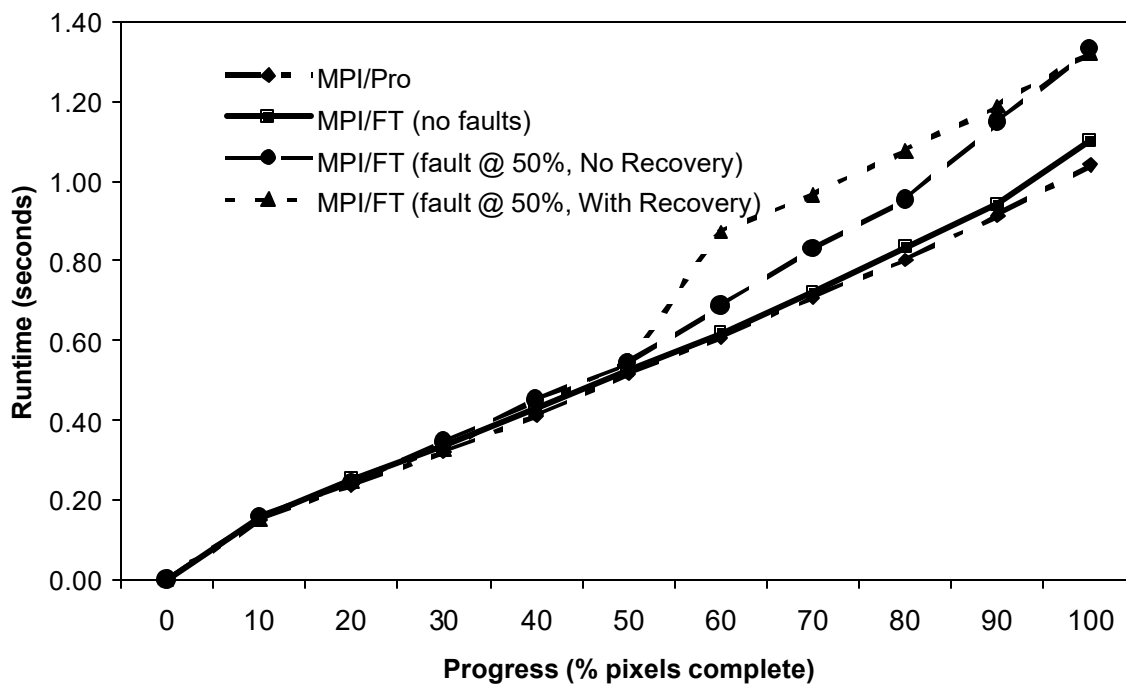


Figure A.4: Pmandel Application Runtime with Fault at 50 % Progress

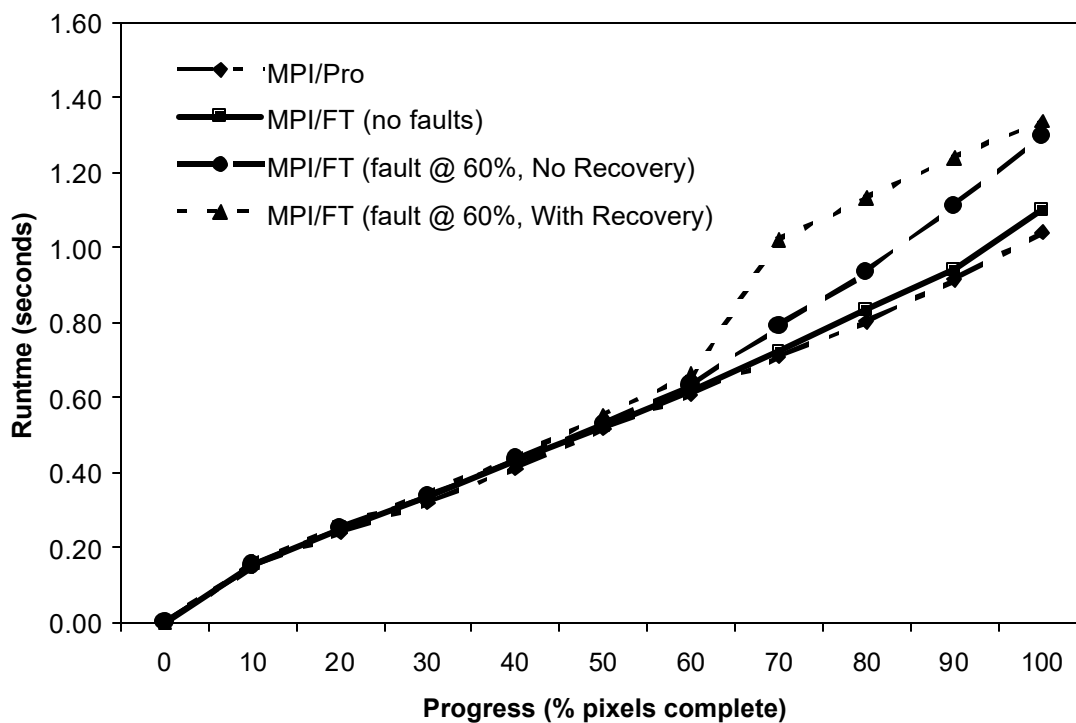


Figure A.5: Pmandel Application Runtime with Fault at 60 % Progress

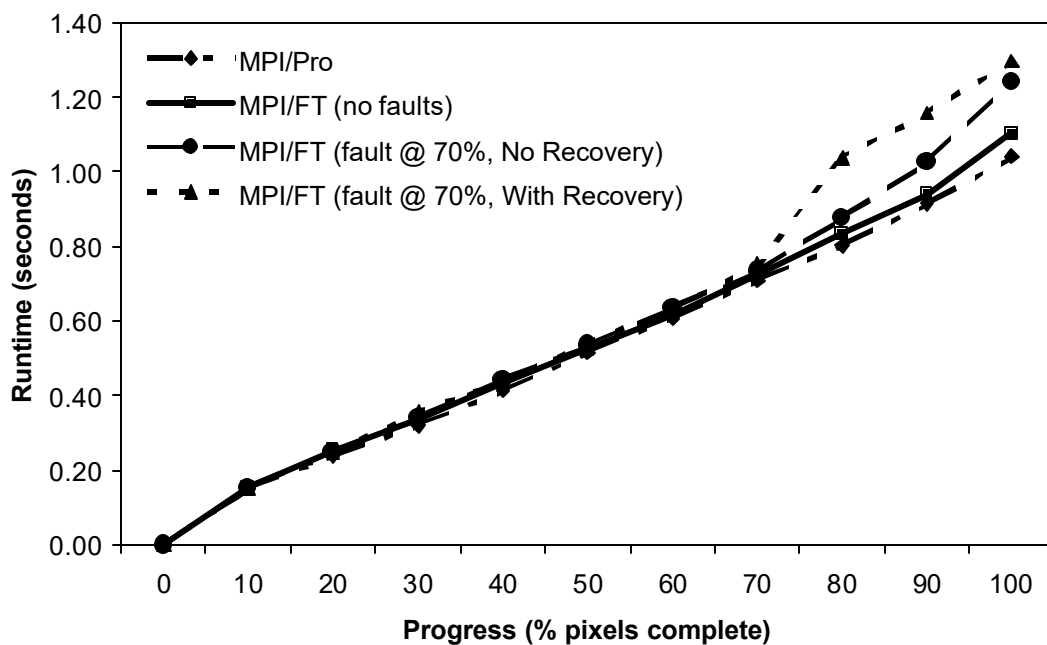


Figure A.6: Pmandel Application Runtime with Fault at 70 % Progress

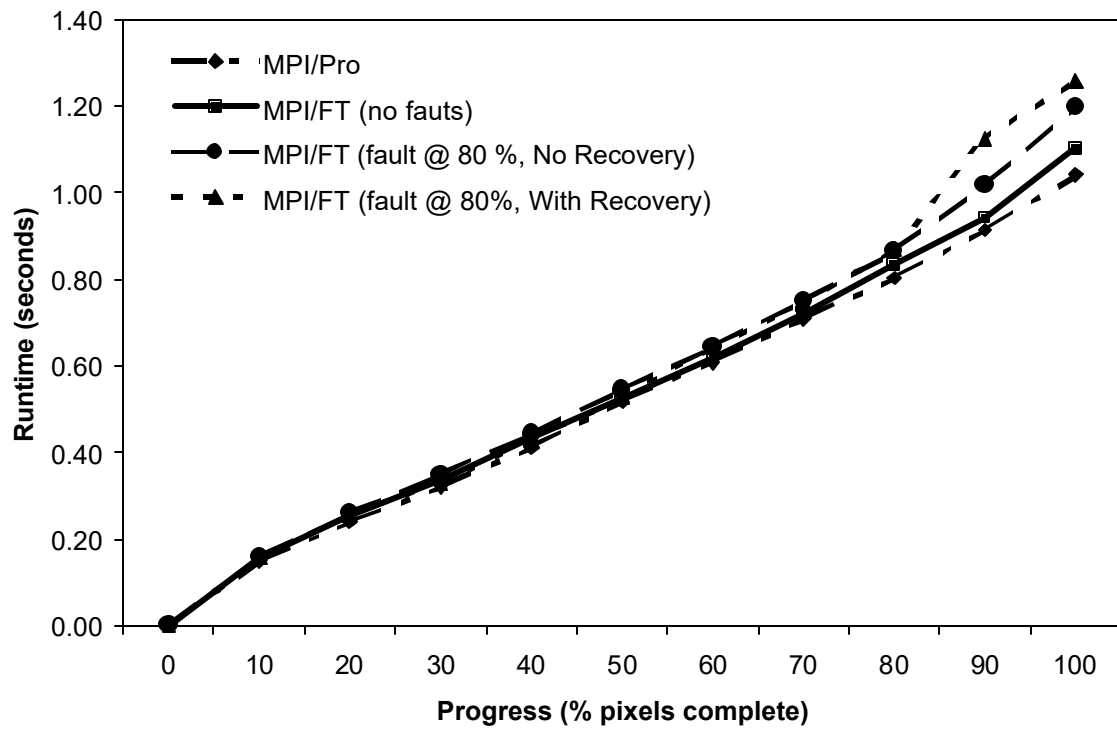


Figure A.7: Pmandel Application Runtime with Fault at 80 % Progress

Appendix B
SCALABILITY TESTS

MPI/FT uses external and internal heartbeats for fault detection. External heartbeats are primarily used between the coordinator thread at rank 0 and SCTs at other processes. This star-based topology between coordinator and SCTs may create a bottleneck and affect the scalability of MPI applications. Experiments and results in this section are focused on measuring the impact of this external heartbeat mechanism on scalability of applications.

Speedup is used as the primary measure to understand scalability of applications. The definition is given by equation B.1,

$$Speedup_{X,NP} = \frac{T_{MPI/Pro,1}}{T_{X,NP}}, \quad (B.1)$$

where $T_{MPI/Pro,1}$ is the runtime for a given application running on MPI/Pro middleware with one process and $T_{X,NP}$ is runtime of the same application running on certain middleware X with NP processes.

The Game of Life [6] program was used to measure the impact of FT mechanisms on scalability. The first runtime results were obtained by running the Game of Life program with varying data grid sizes (16x16, 250x250, 1000x1000) on plain MPI/Pro middleware. These experiments were repeated for varying process sizes (1, 2, 4, 8). The same set of experiments were repeated on MPI/FT middlewares with external heartbeat rates set at 1 Hz and 0.25 Hz, while internal heartbeats were disabled. These tests were performed on a cluster of Intel Pentium machines (900mhz, 768 MB RAM, Linux 2.4 OS). Table 1 presents the speedup values computed from the previous experiments using equation B.1. Figures B.1, B.2, and B.3 present the same information.

Table B.1: Speedup of Game of Life Application with MPI/Pro and MPI/FT

Data Size	Number of Processes	MPI/Pro	MPI/FT	
			EXT HB @ 0.25 Hz	EXT HB @ 1 Hz
16x16	1	1.000	0.909	0.909
	2	0.138	0.137	0.136
	4	0.093	0.088	0.088
	8	0.074	0.070	0.069
250x250	1	1.000	0.970	0.966
	2	1.760	1.752	1.730
	4	2.990	2.919	2.932
	8	3.544	3.465	3.413
1000x1000	1	1.000	0.996	0.996
	2	1.926	1.918	1.917
	4	3.453	3.437	3.431
	8	6.284	6.282	6.280

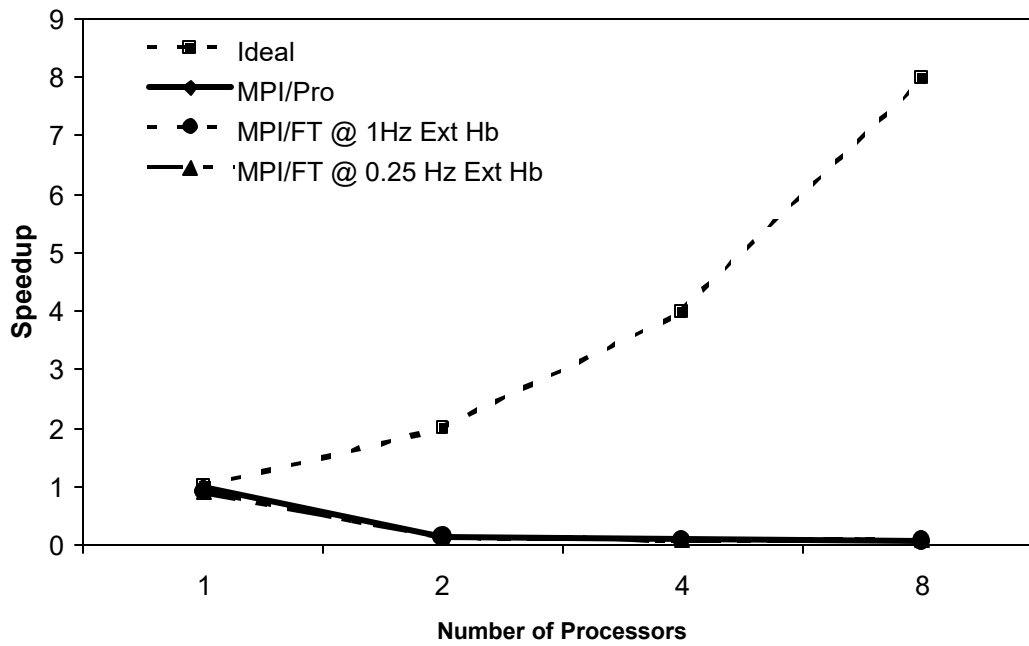


Figure B.1: Speedup of Game of Life, 16x16

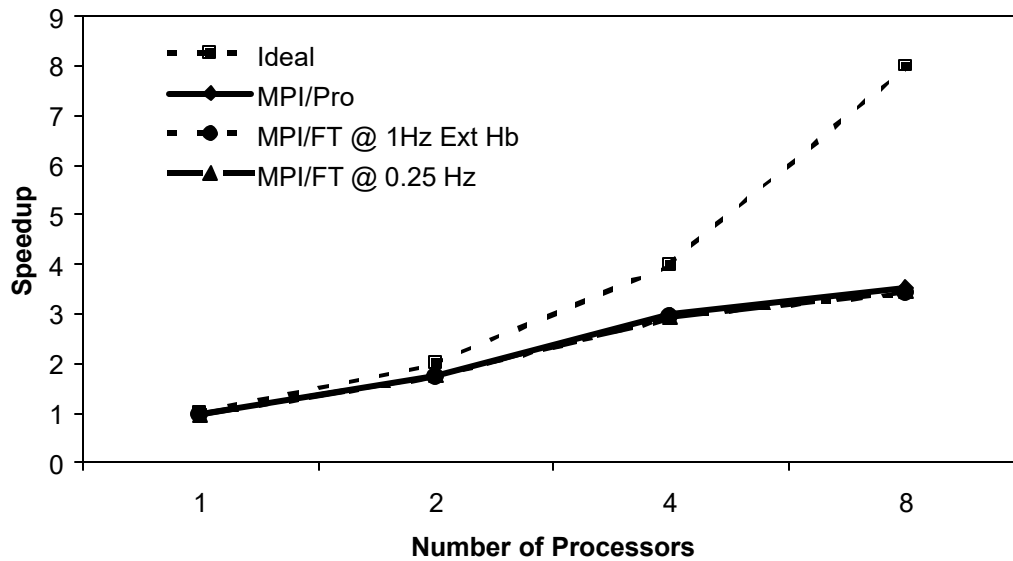


Figure B.2: Speedup of Game of Life, 250x250

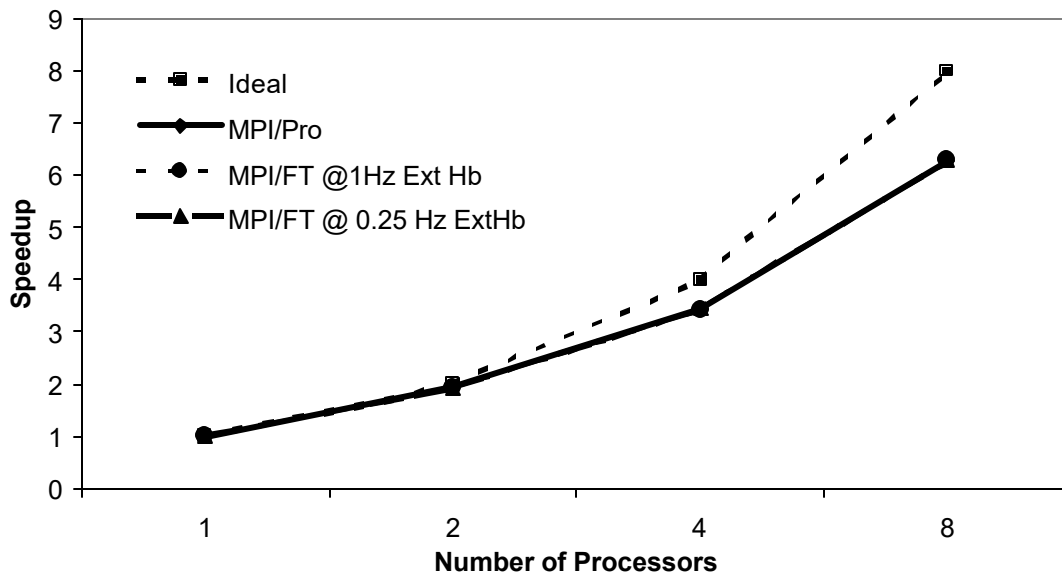


Figure B.3: Speedup of Game of Life, 1Kx1K

Results indicate that the speedup of applications running on MPI/FT middleware is comparable to speedup of applications running on MPI/Pro. The relative difference in speedups is higher for runs with smaller data sizes. This may be attributed to higher impact of MPI/FT mechanisms on communication rather than computation, as discussed in Section 6.1. The scalability of computationally intensive applications is less impacted. However, runs with a larger number of processes are expected to create bottlenecks at rank 0 process, and future work may need to adapt hierarchical and scalable detection mechanisms such as gossiping [10].