A FAULT TOLERANCE PROTOCOL FOR FAST RECOVERY

BY

SAYANTAN CHAKRAVORTY

B. Tech., Indian Institute of Technology, Kharagpur, 2002
M. S., University of Illinois at Urbana-Champaign, 2005

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor Josep Torrellas
Professor Yuanyuan Zhou
Professor Indranil Gupta
Professor Keshav K. Pingali, University of Texas at Austin

Abstract

Large machines with tens or even hundreds of thousands of processors are currently in use. As the number of components increases, the mean time between failure will decrease further. Fault tolerance is an important issue for these and the even larger machines of the future. This is borne out by the significant amount of work in the field of fault tolerance for parallel computing. However, recovery-time after a crash in all current fault tolerance protocols is no smaller than the time between the last checkpoint and the crash. This wastes valuable computation time as all the remaining processors wait for the crashed processors to recover.

This thesis presents research aimed at developing a fault tolerant protocol that is relevant in the context of parallel computing and provides fast restarts. We propose to combine the ideas of message logging and object based virtualization. We leverage the facts that message logging based protocols do not require all processors to rollback when one processor crashes and that object based virtualization allows work to be moved from one processor to another. We develop a message logging protocol that operates in conjunction with object based virtualization. We evaluate and study the implementation of our protocol in the Charm++/AMPI run-time. We use benchmarks and real world applications to investigate and improve the performance of different aspects of our protocol. We also modify the load balancing framework of the Charm++ run-time to work with the message logging protocol. We show that in the pres-

ence of faults, an application using our fault tolerance protocol takes less time to complete than a traditional checkpoint based protocol.

To Ma and Baba

Acknowledgments

I would like to thank my advisor Prof. L. V. Kalé for his encouragement, guidance and patience without which this thesis would have not been possible. His willingness to have long discussions, even in the middle of the busiest day, has helped me and boosted my morale no end. I would also like to thank my dissertation committee Prof. Indranil Gupta, Prof. Keshav Pingali, Prof. Josep Torrellas and Prof. Yuanyuan Zhou for their very helpful suggestions and advice.

This thesis builds on a large body of work by current and former members of the Parallel Programming Laboratory. I am very grateful to Gengbin Zheng and Orion S. Lawlor for their insights and suggestions that helped me out of many a difficult corner. Gengbin has been my tutor in the dark art of debugging parallel programs. Several other members have been of great help by being sounding boards for my fanciful ideas, proof reading papers and acting as another pair of eyes in search of bugs. I would like to thank, in no certain order, Chee Wai Lee, Greg Koenig, Terry Wilmarth, Eric Bohm, Sameer Kumar, Vikas Mehta, Filippo Gioachin, Celso Mendes, Chao Huang, Tarun Agarwal, Yogesh Mehta, Nilesh Choudhury, Amit Sharma, David Kunzman, Isaac Dooley, Abhinav Bhatele, Aaron Becker, Pritish Jetley and Lukasz Wesolowski. They have been excellent company in the lab as well as on many an eating and drinking expedition. Eric and Terry are owed a particular debt for introducing me to a wider variety of beers than I dreamt existed. Chee Wai has also been

a fellow adventurer in search of ever newer cuisines and comedies.

Graduate school was enlivened by a number of fellow travellers. My old room-mates, Rishi Sinha and Smruti Ranjan Sarangi, were always game for a spirited discussion about anything under the sun. I would like to thank Rabin Patra, Devapriya Mallick, Saurabh Prasad, Varij, Nikhil Singh, Swagata Nimaiyar and Divya Chandrasekhar for all the good times.

My parents have been a constant source of support and strength during my work on this thesis. They have seen me through the bad days and laughed with me during the good days. Dad's wakeup calls from half way across the world and Mom's emailed recipes have all greatly helped me along the way. Their responsibility for this thesis extends further back with them never discouraging my incessant *why*s and always encouraging me to think for myself. This would never even have started without them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Massively parallel systems with tens or hundreds of thousands of processors, such as ASCI-Purple, Red Storm and Bluegene/L, are being used for scientific computation. More powerful machines with even larger numbers of processors are being planned and designed. Fault tolerance is important for such large machines. In this thesis, we present a fault tolerance protocol that provides fast recovery from a processor crash. We optimize the basic protocol in various ways to reduce the associated overhead. We evaluate the effects of the optimizations on applications with different characteristics. We also extend our protocol such that it can work along with a dynamic runtime load balancing framework. This lets an application return to optimal performance after a processor failure and recovery.

1.1 Motivation

Scientific computing shows a definite trend towards systems with hundreds of thousands of processors. Many of these systems, built with off-the-shelf components, are likely to suffer from frequent partial failures. Since existing machines already experience failure every few hours [26], future larger machines might have an even smaller *mean time between failure* (MTBF). Therefore, any application running for a significant amount of time on these machines will have to tolerate faults. In fact, applications running on the full Bluegene/L machine

have already faced this problem and have had to incorporate fault tolerance in the application itself to run for long durations on the whole machine [31].

As we shall discuss in Chapter 2, there is a rich set of existing fault tolerance solutions for parallel systems. Some solutions rely on redundancy to provide fault tolerance, such as triple modular redundancy. However, these solutions waste a large fraction of computation power, $\frac{2}{3}$ in the case of triple modular redundancy, even when there are no faults. This extremely high overhead makes redundancy based solutions unsuitable for high performance computing applications where performance is a very important criterion. In almost all other schemes, each processor in an application periodically saves its state (referred to as the processor's *checkpoint*) to stable storage. Stable storage could be a parallel file system or even the local disk of another processor. After a processor crashes its state is recovered from a previous *checkpoint* and all the work on the crashed processor since the previous checkpoint is redone on one processor. So for all these protocols, the time taken for recovery is no less than the time between the previous checkpoint and the crash. We aim to develop a protocol that reduces the recovery time to less than the time between the crash and the previous checkpoint. Such a fast recovery protocol would yield a number of benefits:

- Faster restarts allow us to reduce the overall execution time for an application, when faced with faults. Of course, this depends on the protocol's overhead not cancelling out the benefits of fast restart.

- For most existing protocols, the execution time for an application increases sharply as the fault frequency approaches the checkpoint frequency. This happens because most of the time is spent recovering from faults and very little actual progress is made. On the other hand a fast restart

protocol reduces the amount of time spent in the recovery stage and frees up more time for actually driving the application forward. This allows the application to make progress even when the frequency of faults meets or even exceeds the checkpoint frequency.

- This also means that for a given fault frequency, a fast restart protocol can make progress at the same rate as existing protocols with a higher interval between checkpoints. This helps reduce the total execution time by lowering the amount of time spent checkpointing.

Is it possible to recover faster than the time between the last checkpoint and the crash ? If a processor took a certain amount of time to do a particular amount of work before the crash, it will take about the same amount of time to redo it again after the crash. There is no certain way of lowering the recovery time without reducing the amount of work done by the recovering processor. Therefore, the only effective way of speeding up the recovery of a crashed processor is to *parallelize* the recovery by distributing its work among other processors. However, if the other processors are also rolled back to their previous checkpoints and are busy re-executing, distributing work to them is not going to speed up the restart overall. In such a situation distributing work to them would in fact slow down the restart since the other processors would have more work to complete during recovery. This means that an effective fast restart protocol should never roll back all processors to their earlier checkpoints. Only the crashed processors should be rolled back to their previous checkpoints. This is a vital requirement to enable other processors to help speed up the recovery of a crashed processor.

Moreover, we observed that many scientific parallel applications are tightly coupled. When a processor crashes, some of the other processors, if not all, will

soon stop making progress as they wait for the crashed processor to recover. These stalled processors could be used to speed up the recovery of the crashed processor by distributing its work among them. However, the advantage of parallelizing the recovery is not restricted only to tightly coupled applications. Even for loosely coupled applications, the recovery time can be reduced by parallelizing the work of the recovering processor.

Avoiding rollback on all processors has other advantages as well. When a processor crashes in a weakly coupled application, some processors can continue to execute and make progress while the crashed processor recovers. If we roll back all the processors, not only is a large amount of work on all processors needlessly redone, but also any chance for the application to progress during the recovery is completely wiped out. Unlike weakly coupled applications, strongly coupled applications cannot make significant progress while the crashed processor is recovering. However, it is still useful to avoid rollback on all the processors. Processors save power by not re-executing parts of the application needlessly. This is especially important when one out of a large number (say 10000) nodes has failed. Moreover, with most processors idling, the recovering processor has the interconnect network as well as the file system to itself, thus aiding its faster recovery. It also assures that any data that the recovering processor requests from others is supplied to it quickly. Therefore, we developed a fast restart protocol that does not roll back all processors when one processor crashes.

1.2 Thesis Objectives

The thesis focuses on the development, implementation and evaluation of a protocol that allows a crashed processor to recover much faster than the time

between the crash and the processor's previous checkpoint and without adding significant overhead to the normal no-fault execution of an application. We combine the ideas of object-based virtualization and message logging to create such a protocol. Apart from a fast restart, the protocol has the additional property that it never rolls back a processor that has not crashed. We do not assume the existence of an idealized 'stable' storage that never goes down. Our protocol works within this restriction to handle common failure modes very efficiently, and does not aim for absolute tolerance of all failure scenarios. In particular, the protocol tolerates all single processor failures and most multiple simultaneous processor failures. We increase the overall reliability of a system made out of unreliable components.

As we will show in Chapter 7, a fast restart after a crash can cause a load imbalance among the processors. This load imbalance needs to be rectified to retain the performance advantage of fast recovery. Object based virtualization is helpful for this problem as well since it enables runtime measurement-based dynamic load balancing among processors. Load balancing improves the parallel performance of a number of different applications, particularly on large parallel machines. It is particularly useful for the fast recovery protocol since it can redistribute work after a fast restart. However, as we show later, load balancing negatively affects the reliability of our basic message logging protocol. We augment our message logging protocol so that load balancing does not compromise its reliability while still balancing the work load among the processors in an application.

We implement our protocol by modifying the Charm++/AMPI runtime system. Object based virtualization is the central idea behind the Charm++ runtime system. Since AMPI is a MPI implementation on top of the Charm++ runtime system, our protocol is available to most MPI applications. We evaluate

the benefits afforded by our fast restart protocol. For both Charm++ and AMPI applications, we compare the recovery time for the fast restart protocol to the recovery time for those of other fault tolerance protocols such as log-based protocols without fast restart and checkpoint-based methods. The effect of load balancing on the performance of our protocol is also studied. We also measure the overheads introduced by our protocol and quantify the performance penalty of our protocol for various classes of applications. We characterize the type of applications that are most suitable to our protocol.

1.3 Thesis Organization

Chapter 2 discusses the existing literature for fault tolerance protocols. We evaluate the extent to which existing protocols match the objectives set forth in the previous section. Chapter 3 describes the idea of object-based virtualization and the Charm++ run-time system that implements it. We also describe Adaptive MPI which is an implementation of MPI on top of the Charm++ run-time.

Chapter 4 describes the basic version of our fault tolerance protocol that combines message logging and object-based virtualization. We show that simple message logging protocols do not work in the presence of virtualization, and develop a new protocol that correctly handles the scenarios arising from virtualization. We also provide a proof of correctness for the basic protocol. Chapter 5 talks about the modifications to the basic protocol that are necessary to support recovery from multiple simultaneous faults. It includes an analysis of the improved reliability provided by our protocol along with a proof. Chapter 6 presents the fast restart protocol that can speed up recovery from a crash. It also contains a simple mathematical model to help an user predict if

the fast recovery protocol will be useful for a particular application on a certain machine.

Chapter 7 empirically evaluates the protocol as described till Chapter 6. It shows that the fast restart protocol indeed speeds up the recovery of an application from a processor crash. We also investigate the performance penalty paid by different applications while using our protocol. We also apply optimizations to our message logging protocol that help us reduce the performance penalty paid by different applications while using our protocol.

Chapter 8 demonstrates the need for using load balancing along with our fast restart protocol. We describe the existing load balancing framework in the CHARM++ run-time system and how it was modified to work along with message logging. Chapter 9 evaluates the effectiveness of load balancing in different situations. We also compare the performance of our protocol with that of a traditional checkpoint based protocol. Chapter 9 also includes an evaluation of the memory overhead and optimizations to reduce the memory overhead significantly.

Chapter 11 examines an approach to evacuate a processor where a crash might be imminent. It describes the strategy and the changes to the CHARM++ run-time system required to implement it. Experimental results to evaluate the effectiveness of our protocol are also included.

Chapter 12 summarizes the thesis and highlights the primary contributions. It also points out limitations in our work. Future work that might reduce these limitations is also discussed.

Chapter 2

Related Work

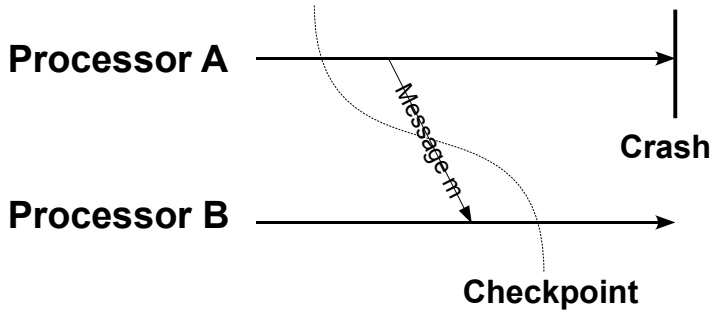
The existing fault tolerance techniques for parallel systems can be divided into two broad categories: checkpoint-based and log-based recovery protocols [22]. There are a number of variations within both categories. All the variations have different pros and cons for our target of providing fast restarts by not rolling back all processors in case of a processor crash.

2.1 Checkpoint Based Protocols

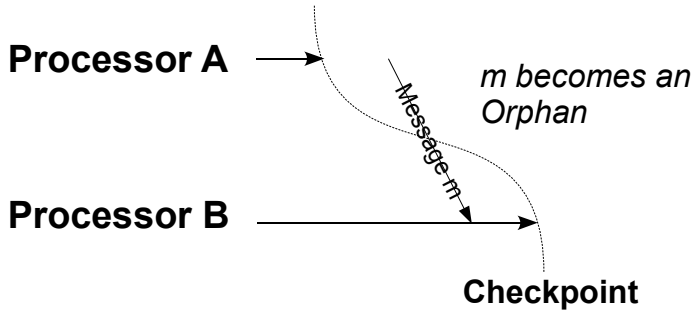
All checkpoint based protocols periodically save the state of a computation to stable storage. After a crash, the computation is restarted from one of the previously saved states. This simple description of checkpoint based protocols belies the fact that the processors in a parallel application need to coordinate to determine the state of the computation at a point in time. The major variants of checkpoint based protocols differ on the degree of coordination between all the processors while taking a checkpoint. There are three major categories of checkpoint based protocols:

- There is no coordination among processors before saving the state of the application in the case of *uncoordinated* checkpointing methods. Each processor saves its state independently of the other processors. The lack of coordination means that checkpoints are fast. Each processor can take its checkpoint when it is most convenient for it (for example, when the

state of the process is small [51]).



(a) Before crash



(b) After crash during recovery

Figure 2.1: Orphan message in an uncoordinated checkpoint

However, uncoordinated checkpointing is usually susceptible to *rollback propagation* during recovery. Rollback propagation is caused by *orphan messages* generated while recovering from a crash. A message is called an orphan when the global state of the computation is such that the message's receiver has received the message but its sender has not sent it. Figure 2.1 shows an example of an orphan message. In Figure 2.1(a) processor A sends message m to processor B. Processor A checkpoints before sending m whereas processor B checkpoints after receiving m. Shortly thereafter, processor A crashes. The two processors are rolled back to their previous checkpoints. At this state of the computation, processor A

has not sent m but B has received m. Thus message m ends up as an orphan. Such a global state is said to be *inconsistent* and can not be used as the starting point of a recovery. If we were to start recovery from this global state, processor A would resend message m and processor B would end up processing it again. Thus, processor B would end up processing message m twice, possibly leading to erroneous results. For a correct recovery, processor B needs to be rolled back to an earlier checkpoint where it has not yet received message m. Of course, rolling back the receiver might make it necessary to rollback other processors, thereby propagating the rollback. Thus uncoordinated checkpoint-based methods not only roll back all processors to previous checkpoints, but the rollback is potentially unbounded as well. This type of recovery makes uncoordinated checkpointing unsuitable to our goal of speeding up the restart time of failed processors.

- All the processors in a computation coordinate to avoid orphan messages and save a globally consistent state in *coordinated* checkpointing schemes. In one of the simplest coordination schemes all processors stop sending messages and wait for all previously sent messages to be received. Once all such messages have been received, all the processors save their state to form a consistent checkpoint [50]. Since there are no messages in transit during a checkpoint, it is guaranteed not to produce any orphan messages. A more efficient non-blocking scheme is Chandy-Lamport's famous distributed snapshot algorithm [18]. The central idea behind this algorithm is the observation that orphan messages can be avoided by storing the messages in transit as a part of the global state of the computation. The coordination mechanism is used to identify exactly which messages

are in transit during a particular checkpoint. Thus, unlike uncoordinated schemes coordinated checkpoints do not suffer from cascading rollbacks caused by orphan messages.

However, coordinated checkpointing forces all processors to rollback to their previous checkpoint when a single processor crashes. Since all processors redo their work since the last checkpoint, recovery time has a lower bound equal to the time difference between the crash and the previous checkpoint. Coordinated checkpoint protocols are the most common fault tolerant technique. A number of implementations provide fault tolerant versions of MPI such as CoCheck [48], Starfish [3], Clip [19] and AMPI [30, 54]. A non-blocking coordinated checkpointing algorithm that uses application level checkpointing is presented by Bronevetsky et. al. [16].

- Communication-induced checkpointing tries to combine the advantages of coordinated and uncoordinated checkpointing. It tries to reduce the cost of checkpointing by allowing processors to take most checkpoints independent of each other. It keeps a cap on cascading rollbacks by forcing processors to take additional checkpoints at certain points in the computation. These checkpoints are taken to periodically create a set of checkpoints (one from each processor) that forms a consistent global state.

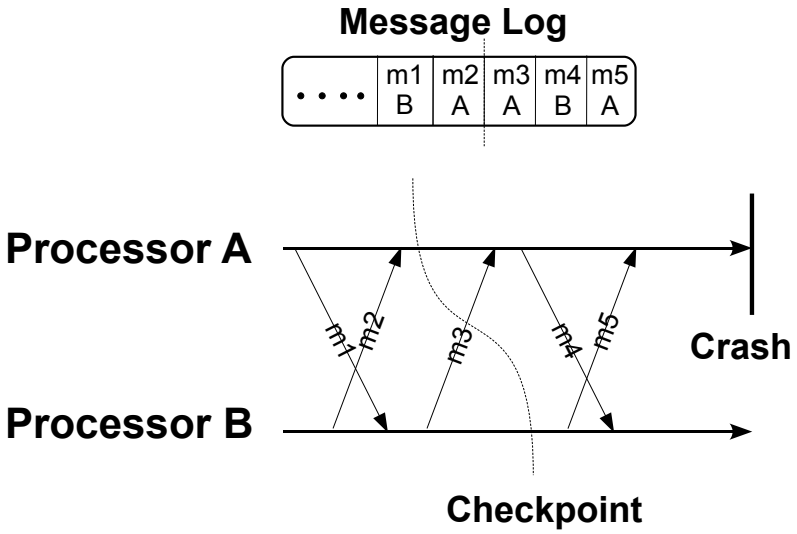
  However it was found that communication induced checkpoint methods did not scale well to large numbers of processors [5] as the large quantity of forced checkpoints nullify the benefit accrued from the autonomous uncoordinated checkpoints. Moreover, like all checkpoint-based methods, during recovery all processors are rolled back to some previous checkpoint. This means that the recovery time is at least equal to the time between the crash and the previous checkpoint. Thus, communication-induced

checkpointing does not meet our criterion of fast restarts.

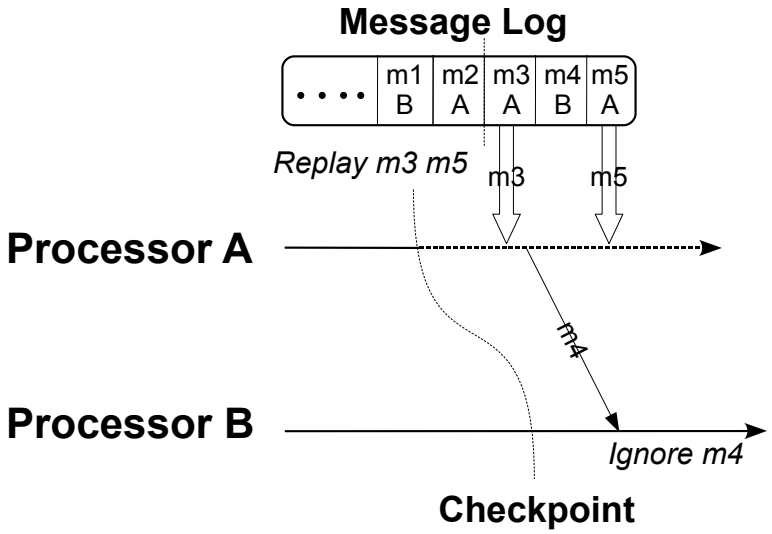
2.2 Message Logging

The second category of fault tolerance protocols are those that depend on stored message logs for recovery. All messages sent during an application are saved in message logs. After a processor crashes, all the messages are resent to the crashed processor and reprocessed in the same order as before the crash. This brings the restarted processor to its exact state before the crash. The piecewise deterministic(PWD) assumption [49] is the key idea underlying all log based recovery protocols. PWD assumes that all the non-deterministic events that affect the state of a process can be recorded in a log along with the data required to recreate those events. All the events are recreated after a crash and reprocessed in the same order as before the crash. This brings the restarted processor to its exact state before the crash. Message logging is based on the observation that messages are the sole source of non-deterministic events in a parallel program. Therefore logged messages can be used to restore the state of a crashed processor after a crash. A message's entry in a message log not only includes the message itself but also the sequence in which it will be processed by the receiver.

Message logging is rarely used in its pure form. Instead it is commonly used in conjunction with periodic checkpoints. Figure 2.2 is an example of such a generic message logging protocol running on 2 processors. Figure 2.2(a) shows the processors periodically take checkpoints in addition to saving messages in message logs. This means that after a processor crashes, an application does not need to rollback to the very start and replay all the messages. As shown in Figure 2.2(b) it rolls back just the crashed processor (A) to its previous



(a) Before crash



(b) After crash during recovery

Figure 2.2: A simple example of a generic message log based fault tolerance protocol

checkpoint and replays the messages A received between the checkpoint and the crash($m3$ and $m5$). Thus, unlike checkpoint based methods, message log based methods do not have to rollback all processors to previous checkpoints.

However, this means that during recovery a processor might re-send messages that have already been received by other processors (like $m4$). Message logging protocols are designed to discard such repeated messages .

The different message logging protocols vary in the way message logs are saved. We now discuss the advantages and disadvantages of the different variants for our goal of a fast recovery protocol. As an aside, we will see that the pros and cons of different message logging protocols have strong parallels with those of the checkpoint protocols.

2.2.1 Pessimistic Message Logging

Pessimistic log based protocols ensure that each received message, including the sequence in which it will be processed, is saved to stable storage before being processed by the receiving processor. If a processor crashes, all the messages processed by it before the crash (messages that might have affected its state) are available on stable storage. So, during recovery it can re-process these messages in the same sequence as earlier to restore its pre-crash state. Any messages that the crashed processor might have sent before the crash are resent during recovery. This means that there can be no orphan messages in the case of pessimistic log based protocols. Therefore pessimistic log based protocols do not suffer from rollback propagation. As a result, each processor needs to store only one checkpoint. Moreover, once the checkpoint of a processor has been saved in stable storage, all the logs of messages processed by it before the checkpoint can be thrown away. This garbage collection of old logs helps keep a cap on the memory overhead of pessimistic message logging protocols.

Pessimistic protocols avoid the problem of rollback propagation by paying the performance penalty imposed by having to save each message's log to stable storage before it can be processed by the receiver. This increases the message

latency and hurts the application's performance.

In spite of this overhead, pessimistic protocols are promising since they do not roll back any processors apart from the crashed ones and meet one of our targets described in Chapter 1.1. Moreover, there are techniques to ameliorate the cost of saving message logs to stable storage. The overhead of logging can be decreased by using specialized hardware [12]. Sender-based message logging [32] is a more interesting and widely applicable method of reducing the performance penalty associated with pessimistic log-based protocols. It is based on the idea that the volatile memory of one processor can act as stable storage for another processor. Therefore, when a processor (say B in Figure 2.2) sends a message to another processor (A), processor B retains a copy of the message in its volatile memory. When A crashes, B can simply re-send its copy of the message. This reduces the overhead of message logging since we do not need to store the message in a central stable storage system. This also reduces the complexity of the system by removing the requirement for any stable storage. MPICH-V1 and V2 [13, 14] are systems contemporary to our work that provide fault tolerant versions of MPI by using pessimistic log based methods. MPICH-V1 is a stable storage based protocol whereas MPICH-V2 is sender based. However, none of the existing systems provide restarts faster than the time between the crash and the previous checkpoint.

2.2.2 Optimistic Message Logging

Optimistic protocols save the message logs temporarily in volatile storage on the receiver before sending them all to stable storage [49]. So, the message latency overhead is significantly lower for optimistic protocols than pessimistic ones. On the other hand, when a processor crashes, the message logs in volatile storage are lost. During recovery, the crashed processor needs these messages

to restore itself to the state before the crash. With the message logs missing, the only way to get those messages again is to rollback the senders and have them re execute so that the messages are generated again. This rolling back of processors other than the crashed one makes optimistic protocols inappropriate for protocols with fast restart. Moreover, garbage collection of message logs in optimistic protocols is also complicated.

2.2.3 Causal Message Logging

Causal logging stores message logs temporarily in volatile storage of the receiver but prevents cascading rollbacks by tracking the causality relationships between messages [21]. Tracking causality and recovery from faults are complicated operations in causal message logging protocols. Manetho[21], MPICH-Vcausal [15] and the protocol by Lee et. al. [40] are examples of causal logging systems.

2.3 Fault Tolerance Support for MPI

Since Message Passing Interface (*MPI*) is the parallel programming paradigm most commonly used by application programmers, most parallel fault tolerant schemes are also implemented for MPI. As the examples cited in the previous two sections show, there is more than one MPI implementation for most of the different fault tolerant techniques. These range from Cocheck, Starfish and AMPI for coordinated checkpoints to MPICH-Vcausal for causal logging. An interesting feature of AMPI is its ability to pack up user data for checkpoints automatically without requiring any user code on some architectures. A protocol that is a hybrid of uncoordinated checkpointing and message logging has also been proposed for MPI [47, 16]. It avoids orphan messages by storing the message logs for messages that are potential orphans. However, like all check-

point based methods it rolls back all processors during recovery after a crash. FT-MPI [23, 24] is a fault tolerant MPI which lets the application handle the rollback and recovery. A detailed discussion about MPI and its relation to fault tolerance can be found in Gropp et. al. [28]. CiFTS (Coordinated Infrastructure for Fault Tolerance Systems) is a recent project that aims to develop a fault awareness and notification mechanism that can be used by different fault tolerance libraries through a common interface [1].

Chapter 3

Object Based Virtualization

Object based virtualization [34] encourages the user to view his parallel computation as a large number of interacting objects. These objects, also known as *virtual processors* (VPs) interact only by sending messages to each other. The user decomposes his application into objects without taking into consideration the number of physical processors. Figure 3.1 shows the user view of such an application in which circles represent objects with the arrows representing messages between them. These objects are mapped to physical processors by the run-time system. The system view shows one such possible where the square boxes represent physical processors containing messages. The run-time system is also responsible for assuring message delivery between objects irrespective of their location.

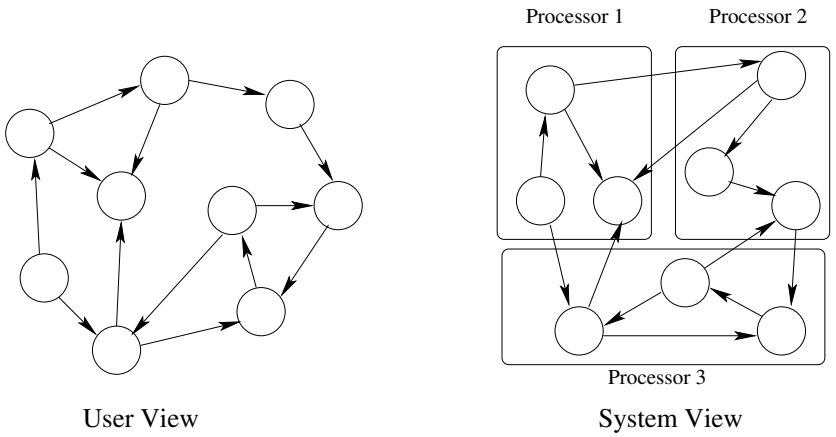


Figure 3.1: The user and system views of an application in Charm++

Object based virtualization enables automatic adaptive overlap between

computation and communication. While an object on a processor is waiting for a message, another object can continue with it computation. Thus, the communication and computation of different objects on a processor can be overlapped without any extra programming effort on behalf of the user. This makes the application more latency tolerant.

Virtualization also lets the runtime system perform measurement based dynamic load balancing. Since the run-time system is solely responsible for mapping objects to physical processors, it can also change the mapping during execution by migrating objects from one processor to another. Moreover, since the runtime system is responsible for all message delivery it can accurately track the communication and computation load of different objects. As most scientific applications are iterative and show good persistence of communication and computation loads over time, the measured load data acts as a good predictor for loads in the immediate future. So, the run-time system can use the load data to calculate a new mapping of objects to processors that provides a good computation and communication load balance across processors. Such periodic load balancing is very useful for obtaining good performance in applications whose load characteristics change with time. Run-time load balancing is also useful because it frees the user from having to load balance his code himself. Instead all he needs to do for a good load balance is to decompose his computation into a large number of objects.

3.1 Charm++

Charm++ [33] is a C++ based object oriented message driven parallel programming language based on the idea of object based virtualization. The virtual processors are C++ objects that send messages to each other through

asynchronous method invocations. Charm++ arranges the virtual processors into collections that can have user defined indices. The underlying Charm++ run-time system supports efficient message delivery between the virtual processors even when some of them are migrating from one physical processor to another [39]. Charm++ also supports collective operations such as broadcasts, reductions and multicasts over these collections of virtual processors.

Charm++ also has an elaborate load balancing framework. It includes instrumentation code to accurately capture the communication and computation load of different virtual processors. This instrumentation code can be turned on and off at the discretion of the user. The framework can be used to collect this instrumentation data from all objects on all processors to come up with a mapping of objects to processors. A large number of strategies have been implemented to calculate this mapping. Different strategies have been found to be suitable for different applications. The load balancing framework is discussed in greater detail in Chapter 8.

Charm++ has been used to develop a number of highly scalable application codes like NAMD for molecular dynamics [44, 35], Changa for computational cosmology [27], leanCP for quantum chemistry [11] as well as frameworks such as ParFUM for unstructured mesh applications [38, 42].

3.2 Adaptive MPI

Adaptive MPI (*AMPI*) is an implementation of MPI on top of Charm++. Each MPI process is implemented as a user level thread, with many such threads on one physical processor. Every user level thread is associated with a Charm++ object and such a pair is referred to as an AMPI virtual processor. In an AMPI VP the user MPI code runs in the context of the user level thread. Blocking

MPI calls are implemented by simply suspending the user level thread. This blocks the user code running within the VP that made the blocking call, while leaving the physical processor free to run user code from other VPs. This allows a user to adaptively overlap communication and computation even while using blocking calls. In a traditional MPI, such overlapping would require the use of non-blocking calls, frequent probing and a more complicated algorithm. Most MPI codes can be run using AMPI with very minor changes.

All the communication of an AMPI VP is handled by its Charm++ object. The Charm++ object not only is responsible for simple point to point messaging but also collective operations like broadcasts, multicasts, scatter-gather etc. Since a Charm++ object performs all the communication, AMPI can make use of communication libraries within Charm++ that optimize different patterns of communication among Charm++ objects on multiple physical processors.

The user level threads used in AMPI are migratable and can be moved among different physical processors. This allows an AMPI VP to migrate to another processor by moving both the user level thread and the associated Charm++ object to that processor. Thus AMPI can take advantage of the Charm++ load balancing framework to perform dynamic measurement based run-time load balancing. We augment the MPI specifications with a **MPI_Migrate** function. **MPI_Migrate** is a collective blocking call that can be used to perform load balancing among all the AMPI VPs in an application. The user calls **MPI_Migrate** to show exactly when in an application he wants load balancing to occur.

Chapter 4

Combining Message Logging and Object Virtualization

Our fault tolerance protocol is entirely software based and doesn't depend on any specialized hardware. However, it makes the following assumptions about the hardware:

- The processors in the system are fail-stop [46]. It means that when a processor crashes it remains halted and other processors may detect its crash. This is a reasonable assumption for common failure modes on parallel machines.
- All communication between processes is through messages.
- The PWD assumption should hold. It is assumed that the only non-deterministic events affecting a processor are message receives.
- No event other than a fault interrupts the processing of a message.
- The machine is assumed to have a system for detecting faults.

We bring together the ideas of sender based pessimistic message logging and object based virtualization to develop a fault tolerance protocol that provides fast recovery from faults on a machine that meets the above assumptions. As mentioned in Chapter 2.2 a sender side message logging protocol stores a message and the sequence in which it will be processed by the receiver on the sender. This reduces the overhead of pessimistic message logging and also removes the need for an idealized stable storage. Virtualization also affords us a number

of potential benefits with respect to our message logging protocol. It is the primary idea behind faster restarts since it allows us to distribute the work of the restarting processor among other processors. The facility of runtime load balancing can be utilized to restore any load imbalances produced during fast recover. Virtualization also makes applications more latency tolerant by overlapping communication of one object with the computation of another. This can help us hide the increased latency caused by the sender side message logging protocol.

Although a synthesis of virtualization and message logging can lead to significant advantages, there are considerable challenges in combining the two. One major problem occurs when dealing with virtual processors on the same processor sending messages to each other. We discuss this in greater detail in Chapter 4.1.2. Similarly moving objects from one processor to another during load balancing also introduces additional complications. That issue is discussed in Chapter 8.

We combine the ideas of virtualization and message logging by treating the virtual processors, and not the physical processors, as the communicating entities that send and receive messages. Since an object's state is modified only by the messages it receives, we can apply the PWD assumption to virtual processors instead of physical processors. After a crash, if a virtual processor re-executes messages in the same sequence as before, it can recover its exact pre-crash state. So, we run the sender based message logging protocol with the objects as participating entities instead of physical processors.

Our design of a pessimistic sender based message logging system that works alongside processor virtualization has three major components: message logging, checkpointing, and restart. Although all three components are very closely related, we describe them as separate protocols for the sake of clarity. As we

shall see, processor virtualization has a significant impact on all components. We discuss the protocol for single faults in this chapter. We extend it to deal with multiple faults and perform fast restart in Chapter 6.

4.1 Message Logging Protocol

We designed the message logging protocol such that during recovery after a crash:

- A Charm++ object (or AMPI thread) on the crashed processor processes the same messages in the same order as before the crash.
- A Charm++ object on other processors does not process a message that it has already processed.

In order to meet these objectives, an object establishes a strict ordering among the messages it receives and processes them in that order. We store both the messages and the order in which they are processed on the sender side. A receiver also keeps track of the number of messages it has processed and the ordering established among those messages. We do some additional book keeping for messages between objects on the same processor.

The protocol augments each Charm++ message sent between objects with four data fields as depicted in Figure 4.1 :

1. The unique *id* of the object that sent this message, that is *sender id*. In the example in Figure 4.1 the sender is object α.
2. The unique *id* of the object that is to receive this message, that is *receiver id*. The receiver in the example is β.

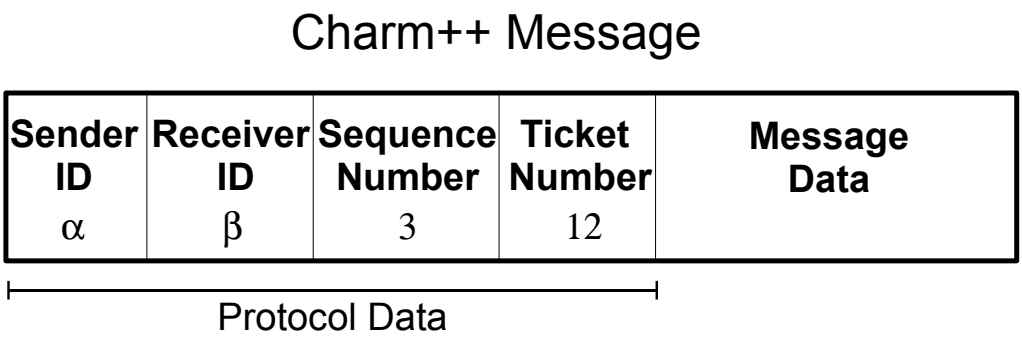


Figure 4.1: Protocol data added to a Charm++ message

3. The *sequence number*(*SN*) of this message. The SN is a count of the number of messages sent from the message's sender to its receiver at the time this message was generated. In the example this is the 3^{rd} message sent by object α to object β.

4. The *ticket number*(*TN*) of this message. The receiver of a message assigns it a TN. An object processes all the messages it receives in increasing order of their TNs. TNs for all the messages received by an object form a single contiguous sequence. The message in the example will be the 12^{th} processed by object β.

As seen in Figure 4.2 each Charm++ object maintains a number of data fields and tables:

1. Each Charm++ object is given a unique *id*, for example α in case of the object shown in Figure 4.2.

2. An object stores the highest TN processed by it as *TNProcessed*. In the example object α has already processed 17 messages.

3. An object stores the highest TN assigned by it as *TNCount*. Object α has assigned ticket numbers to 19 distinct messages.

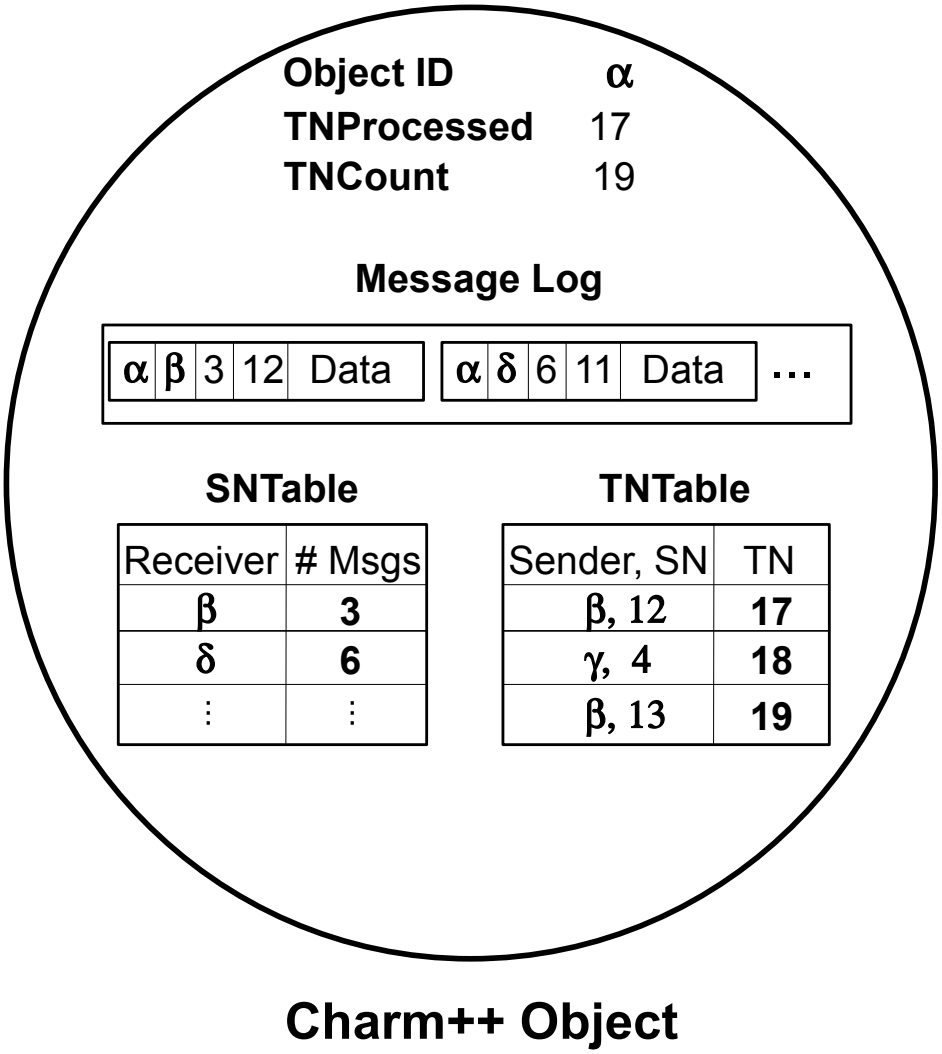


Figure 4.2: Protocol data added to a Charm++ object

4. Each message sent by an object is stored in the object's *message log*. The example shows two messages in the message log of α sent to objects β and δ. The protocol data for each message, namely sender ID, receiver ID, SN and TN, are stored along with the message.

5. Every object maintains a table called the *SNTable* for the number of messages sent to different objects. It is indexed by the receivers' id. In the example α has sent 3 messages to β and 6 to δ.

6. An object stores the sender's id, SN and TN for each message received since the last checkpoint in a table called the *TNTable*. The TNTable is indexed by the sender's id and SN of the message. In the example α assigned TNs 17 and 19 to two message with SN 12 and 13 respectively from β. A message with SN 4 from γ was alloted a ticket number 18. As described in Section 4.1.1, this table is used by an object to assign TNs to all messages destined for it.

4.1.1 Remote Mode

When the sender(α) and receiver(β) objects are on different processors, the message logging protocol is said to operate in the *remote* mode. Figure 4.3 illustrates an example in which object α sends message m to β on another processor. Before sending a message, α looks up β in its SNTable and finds the number of messages sent to β previously. Object α increments that count and assigns it as the SN of the message. In the example in Figure 4.3 the message m gets a SN of 4. Object α then stores the message in its message log. As seen in Figure 4.3, the sender α then sends a request for a ticket, consisting of α's id and the message's SN, to β.

On receiving the request, β looks up the tuple $\{\alpha, SN\}$ in β's TNTable. If there is no matching entry in the TNTable, there are two possibilities: i) the common case that α is sending a new message to β, ii) the rare case where the sender α is recovering from a fault and is re-sending a message that was processed by β before its last checkpoint. We can distinguish between the two cases by comparing the SN in the ticket request to the lowest SN from α in β's TNTable (say l). If the SN in the ticket request is higher than l, then we are dealing with the common case that α is sending β a new message. Object β increments TNCount and decides on this value as the TN. β also adds an entry

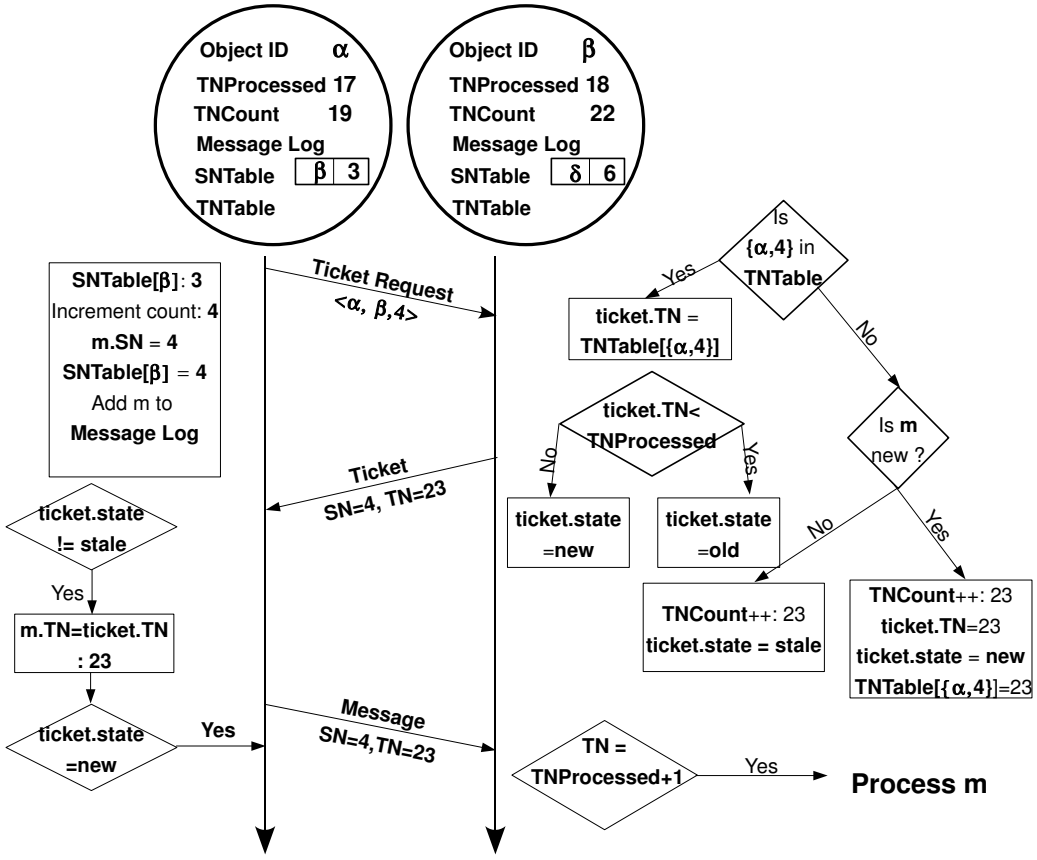


Figure 4.3: Remote mode of the message logging protocol

for the tuple $\{\alpha, SN, TN\}$ to its TNTable. The TN is returned to the sender α along with α's id and the SN. The example in Figure4.3 is this common case and β allots it a TN of 23.

If the SN in the ticket request is lower than l then the sender α has sent a ticket request for a message that was processed by β before its last checkpoint. Since pessimistic message logging does not suffer from cascading rollbacks, β can safely tell α to discard this message by labeling the ticket as *stale*.

If β finds an entry corresponding to the ticket request in its TNTable, it means that in the past β has assigned a TN to this message from α and that α is recovering from a fault. Object β will reply back with this TN. If the value of this TN is lower than the TNProcessed for β, β marks the TN as *old*. An old TN corresponds to a message β has already processed since the last checkpoint.

When α receives a ticket in reply, it checks if the ticket is stale. If it is not it assigns the TN to the message stored in its log. As Figure 4.3 shows, if the received ticket is marked as new, α sends the message to β.

When β receives the message, it checks if the message's TN is less than or equal to its TNProcessed. If it is, β discards the message as it has already processed this message and should not do so again. If the message's TN is higher than $TNProcessed + 1$, β defers processing this message. If the message's TN is exactly equal to $TNProcessed + 1$, then β processes the message and then increments β's TNProcessed by 1.

The time between the sender starting to send a message and the receiver sending a message of its own as a result of processing the sender's message is increased by the the round trip time of a short message. This is the same as in the sender side message logging protocols of [32, 14]. However, as we shall in Chapter 7, virtualization allows us to mitigate the penalty imposed by this increased latency.

4.1.2 Local Mode

If we were to use the above protocol for messages between two objects on the same processor, the log of a message would exist on the same processor as its receiver. If this processor crashes, all record of the TN allotted to the message would be lost. It would become impossible to re-execute the messages in the correct sequence. Therefore we define a *local* mode of the message logging protocol to deal with this case.

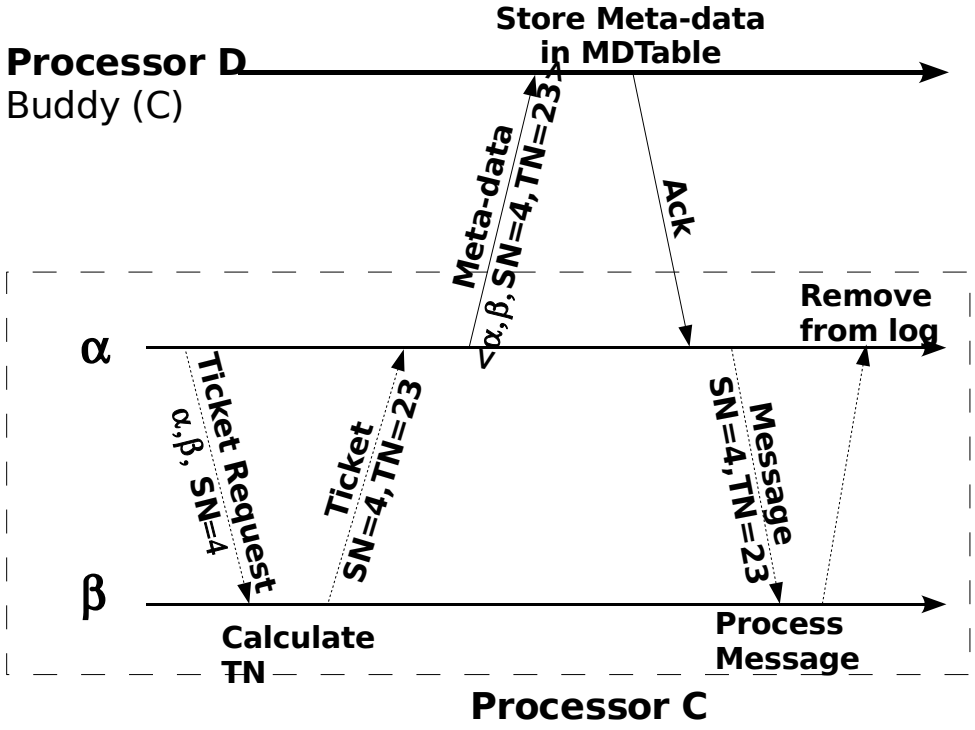


Figure 4.4: Messages in the local mode of the message logging protocol

Each processor is assigned a *buddy* processor. A processor has only one buddy and is the buddy of only one processor. Let us say that object α on processor C wants to send a message to object β on the same processor. As the first step, α assigns the message a SN in the same way as in the remote mode described in Chapter 4.1.1. As Figure 4.4 shows object α then asks β for a ticket

by invoking the ticket generation routine with α's id and the message's SN as arguments. The ticket generation routine uses the same algorithm described in Chapter 4.1.1. We are able to use a method invocation instead of a message because in this case, α and β are on the same processor. The example message in Figure 4.4 is allotted a SN of 4 by α and a TN of 23 by β.

After β has returned a ticket number, α sends the *message meta data* to the buddy of processor C, namely processor D. The message meta data consists of the tuple sender's id, receiver's id, SN and TN of the message. For the example, the message meta data is the tuple $< \alpha, \beta, 4, 23 >$. Processor D, buddy of processor C, stores the meta data for the message in the *message meta data table*(*MDTable*). A processor stores the meta data for messages between objects existing on it, in the MDTable maintained on its buddy processor. After processor D has saved the meta data for the message in its MDTable, it sends an acknowledgment to object α on processor C.

Object α sends the message to β only after receiving this acknowledgment from D that the meta-data for the message has been stored in D's MDTable. As a result, the latency for a message to a local object becomes the same as that of a message to a remote object. After β has processed the message, it tells α to remove the message from its message log. Thus a local message need not be maintained in the sender's log once it has been processed by the receiver. So, the memory overhead of local messages is less than that of remote messages. We can throw away the message log after β has processed it because, if processor C crashes later on the log of the message on α will disappear with the crash. α will not be able to simply re-send the message to β from its log. It will have to re-execute from the previous checkpoint and re-generate the message. So, keeping the message in α's log once β has processed it is useless and just wastes memory. The reason why the log must be maintained till β has processed it is

explained in Chapter 4.3.

4.2 Checkpoint Protocol

The checkpoint protocol not only stores an object's checkpoint but also provides a mechanism to perform garbage collection of the message logs. The state of a Charm++ object consists of user data, a small amount of runtime system data as well as TNCount, TNProcessed, SNTable and the messages in the message log that were sent to objects on the same processor (the reason for this is explained in Section 4.3). Since messages to local objects are deleted from the sender's log once they have been processed by the receiver, only those local messages that have been sent but not processed by their receivers are part of an object's checkpoint. All objects on a processor are checkpointed at the same time.

Figure 4.5 shows an example in which processor C, containing two objects α and β checkpoints. Before the checkpoint, object δ sends message $m1$ to β, while γ sends message $m2$ to α. Objects γ and δ both exist on processor E. Object β also sends a message, $m3$ to α on the same processor. The meta-data tuple, consisting of sender, receiver, SN and TN, for each message is shown in Figure 4.5. All the messages are processed before processor C checkpoints. Processor C packs up the state of all the objects on it and sends it to its buddy, processor D. Each object on C also stores its TNProcessed at the time of checkpoint as *TNCheckpointed*. In the example α gets a TNCheckpointed of 17 and β gets 18. D stores the new copy of C's checkpoint, deletes the old copy and sends an acknowledgment to C. On receiving the acknowledgment, the TNTable of each object on C can garbage collect entries with TN less than TNCheckpointed. Although not shown in the figure to avoid cluttering, α removes $m2$ and $m3$ from its TNTable, while β removes $m1$. Each object

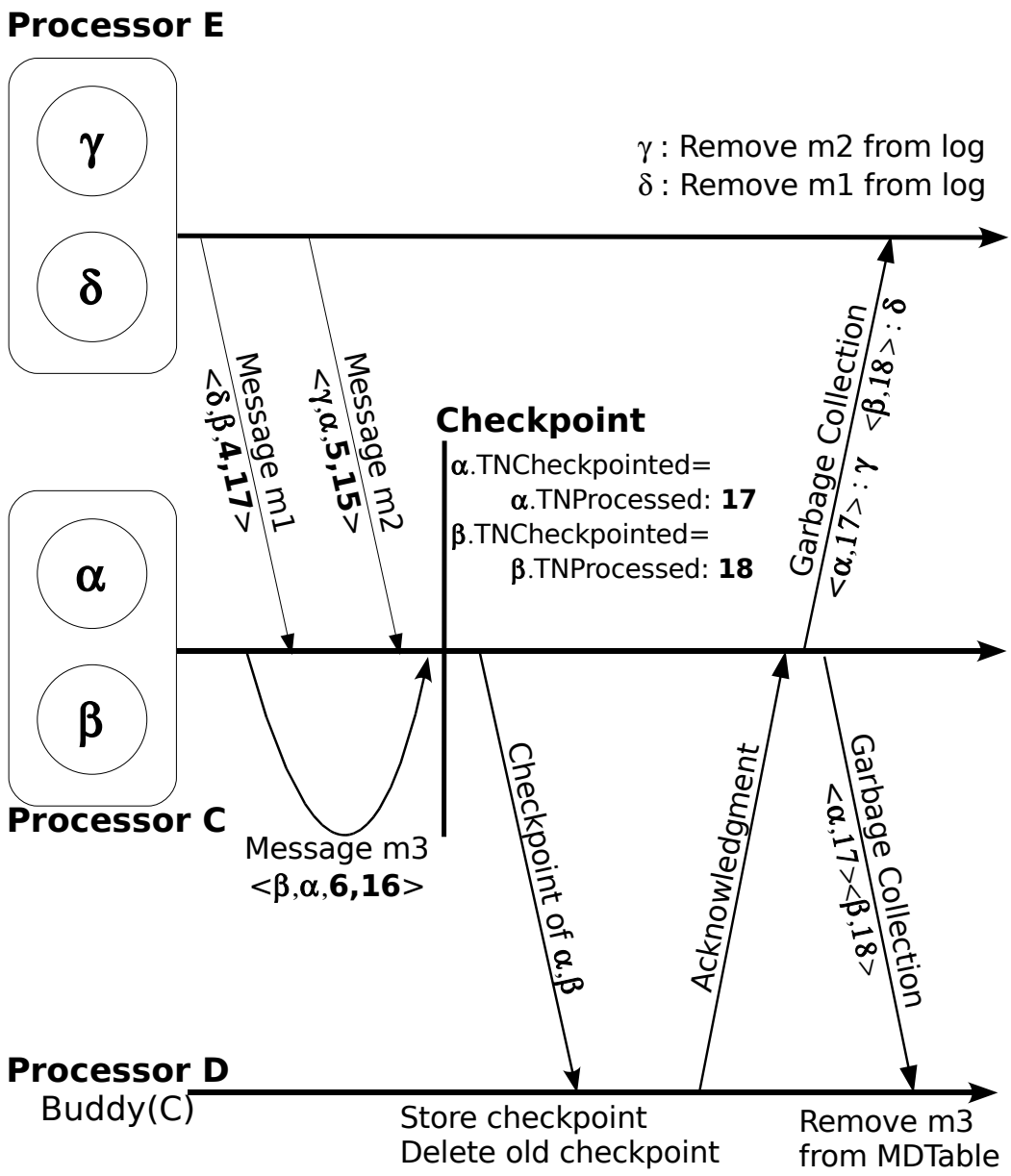


Figure 4.5: Messages in the checkpoint protocol

on C sends out garbage collection messages containing TNCheckpointed to all objects that had sent it messages since its previous checkpoint. If we send out a separate garbage collection message from each object on processor C to all its senders, we will end up sending a large number of tiny messages. We avoid this by consolidating all the garbage collection messages from all objects

on processor C to all objects on another processor into one garbage collection message. As a result, processor C sends only one garbage collection message to each processor that has at least one object that sent a message to an object on processor C. In the example processor E receives a combined garbage collection message: α with TNCheckpointed 17 for γ, β with TNCheckpointed 18 for δ.

When a processor receives a garbage collection message, it gives each of its object the relevant portion of the garbage collection message. When γ receives the garbage collection message from object α it removes all messages to α in its message log that have a TN lower than the TNCheckpointed. In the example γ removes $m2$ from its message log, while δ removes $m1$. A similar garbage collection message is sent to processor D, so D can remove old entries from the MDTable. Figure 4.5 shows processor D removing the entry for message $m3$ from its MDTable. Garbage collection is done lazily so that it interferes as little as possible with the application. An important point about the checkpoint protocol is that it is never blocking. Objects are free to process messages while the checkpoint protocol is ongoing.

We have to deal with an interesting trade off between memory and speed while deciding when to checkpoint. If the checkpoint period is too low, the message logs on senders are garbage collected frequently. This saves memory but increases the time cost because of frequent checkpoints. If the period is too high, the message logs on senders become large though the checkpointing cost is lower. The rate of expected failure is also an important factor in deciding the checkpoint period. Checkpoints might also be performed when the message logs become larger than a particular size.

Storing the checkpoint in memory is not a problem for applications with a small checkpoint state such as molecular dynamics. However, if the application is memory intensive the checkpoint can be stored on the local disk of the buddy

processor. If there are no local disks in the system, the checkpoint can be stored on the cluster's file system. Even message logs can be lazily moved to local disk or the file system to keep the memory overhead low. Of course, moving checkpoints and message logs to disks from memory will slow down restart.

4.3 Restart Protocol

In this section, we describe the basic restart protocol. We illustrate how it works with the help of an example. Figure 4.6 shows the messages received by two objects α and β on processor C after a checkpoint. α receives messages $m4$, $m6$ and $m7$ from γ, β and δ respectively. β receives message $m5$ from δ. The meta-data for each message is shown. After all the messages have been processed processor C crashes.

We assume that a pool of spare processors is available to the parallel job. When the crash detector finds out that processor C has crashed, it restarts a Charm++ process on a spare processor. Figure 4.7 shows the messages exchanged after the new processor C has started up. Processor C asks D for its checkpoint and MDTable. C recreates all the objects that used to exist on it (α and β in the example) from the checkpoint fetched from D. The entries in the MDTable are separated by receiver and added to each receiver's TNTable. C then broadcasts a request to resend logged messages. The request to resend logged messages contains the id and TNProcessed at time of the checkpoint for each object on C.

When a processor, like E, receives a request to resend logged messages, each object resident on it looks in its message log for messages sent to the objects recreated on C. If such a message has a TN more than the TNProcessed for the recreated object it is resent. In Figure 4.7 γ resends $m4$ to α while δ resends

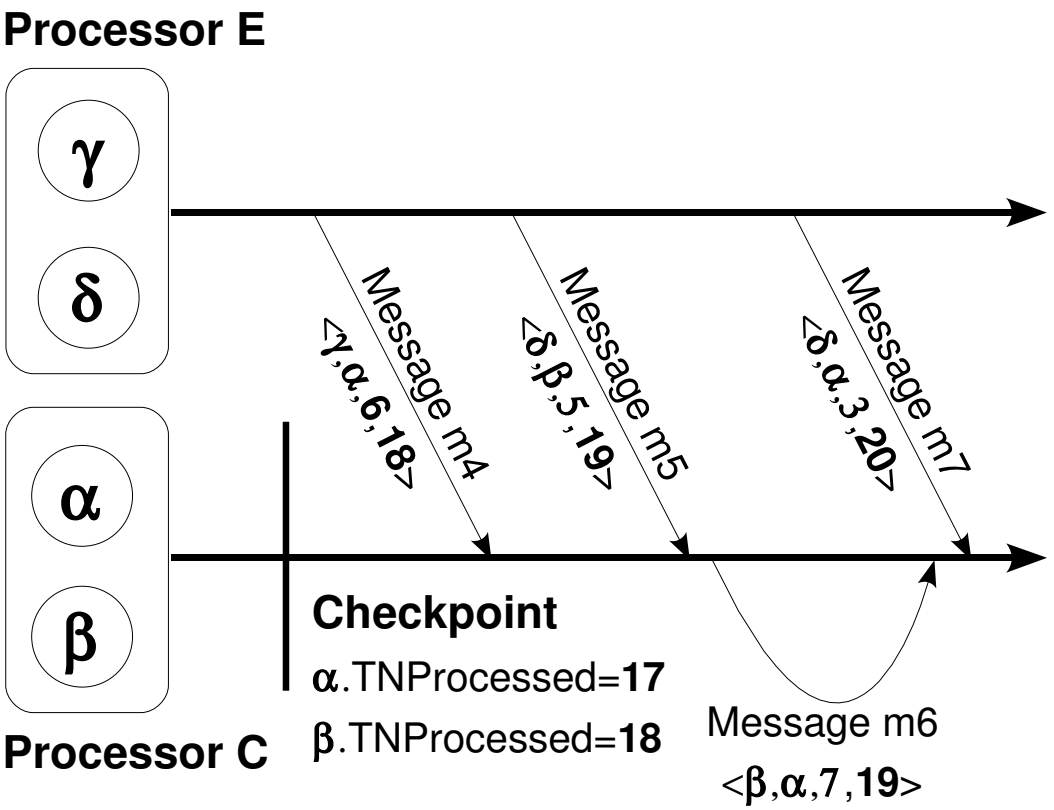


Figure 4.6: Messages before a crash

$m5$ and $m7$ to β and α respectively. Message $m6$ sent by β to α is regenerated during β's re-execution. α looks in its TNTable and finds the entry for $m6$ that was added from the MDTable fetched from D during the re-creation of α. Thus $m6$ gets its old TN 19.

While resending if an object finds a message in its log, that is destined for an object on the restarted processor but does not have a ticket, a new ticket request is issued for that message. However, a restarted object like α must not only give messages the same TNs after a crash as before it but also make sure that no TN is skipped. α should not skip handing out any TN since α will not be able to process any message with a TN higher than the skipped one. Object α on processor C collects a list of the TNs of all the messages resent to it. α then adds to this list the TNs of messages in the MDTable obtained from C's

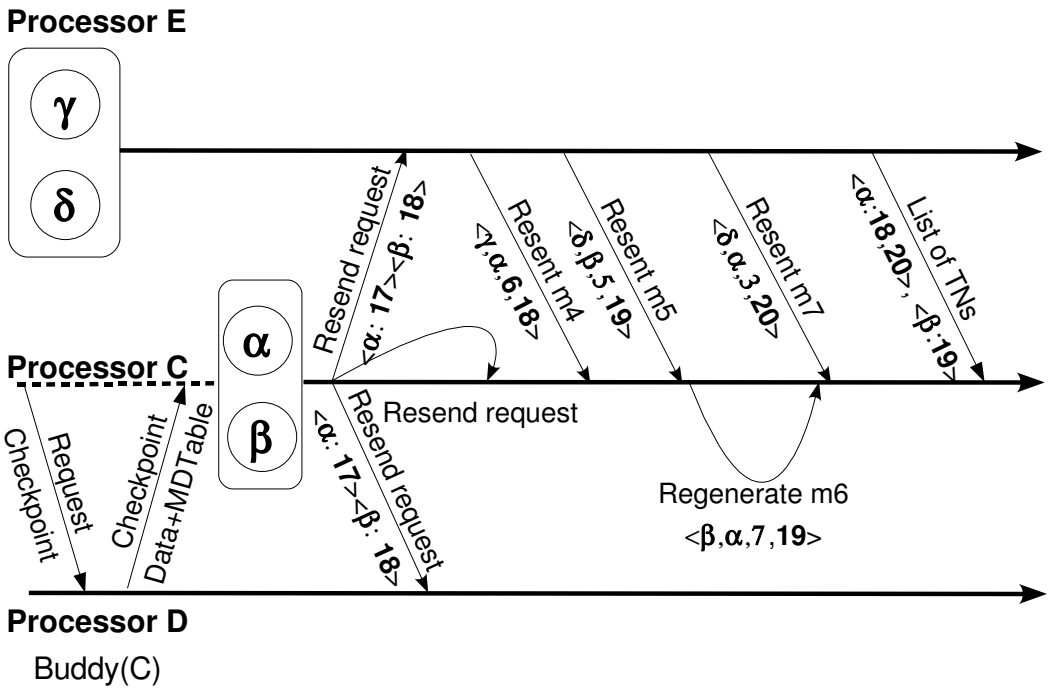


Figure 4.7: Messages during a restart

buddy D. This list of TNs is then sorted in increasing order. After sorting this list α might find that some TNs in the middle are missing. These missing TNs correspond to messages that were given TNs but were not processed by α before the crash. We are sure they were not processed since α would have processed a message only after its TN had been saved in either the sender's log or the MDTable. So when α gives out new TNs it hands out these missing TNs before continuing with TNs higher than TNCount. Each processor combines the list of TNs sent to different objects on the restarted processor into one message. Thus in Figure 4.7 processor E sends a message containing the TNs of messages sent to α as well as β by objects on processor E.

Figure 4.8 shows a special case that the earlier example does not. Figure 4.8 shows the situation during the forward path. Object α sends message $m8$ to object β also on processor C. As the first step α saves $m8$ in its message log. According to the local mode of the message logging protocol the meta-data

for $m8$ needs to be saved in the MDTable of C's buddy processor D before $m8$ can be processed. However, before the acknowledgment for $m8$ can come back from D, processor C checkpoints. Since the checkpoint of an object includes messages in its log sent to other objects on the same processor, α includes $m8$ in its log as part of its state.

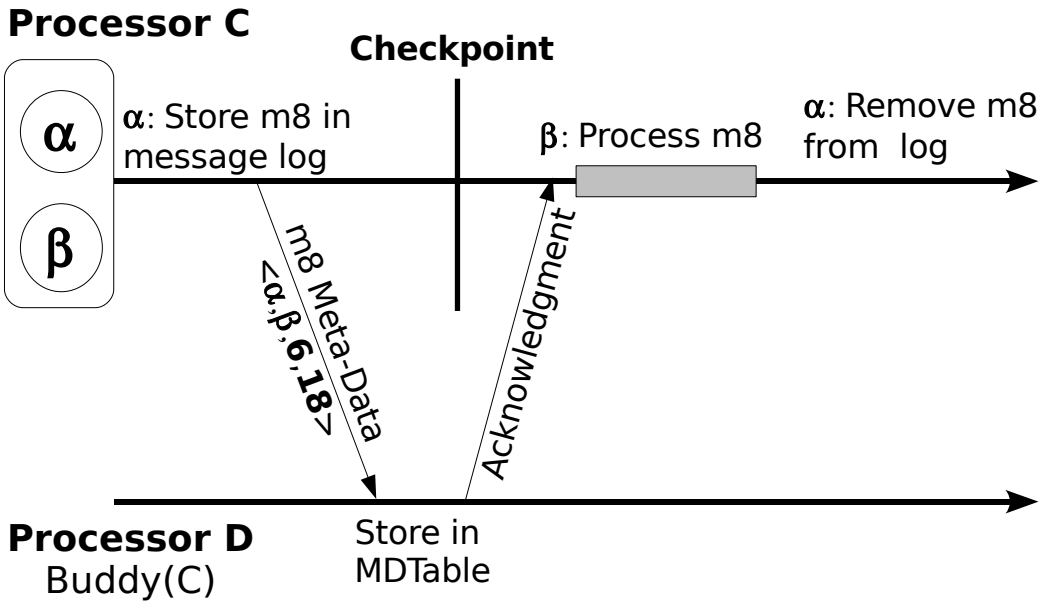


Figure 4.8: A special case for the restart protocol.

When processor C crashes later, it is rolled back to this checkpoint. In this state, α will not regenerate $m8$ since it was sent before the checkpoint and we never rollback beyond one checkpoint. So, α needs to resend message $m8$ for β to recover correctly. Since $m8$ is present in α's message log as part of its checkpoint, α can easily resend the message to β. Therefore during restart, not only objects on other processors but also objects on the restarted processor need to resend messages in their logs.

4.4 Proof of Correctness

Our fault tolerance protocol, like all other such protocols, seeks to guarantee that after recovery from a crash the state of the computation is *globally consistent*. Alvizi et al. provide a precise specification of the global consistency property [6] in the context of message logging protocols. It states that at the end of recovery, there should be no *orphan processors*. This means that after recovery is complete there should be no processor which has processed a message that has not been sent by any processor. This amounts to saying that there should not be any *orphan messages*, as defined in Chapter 2.1, in the system once recovery is complete. We extend this definition to objects by saying that there should not be any *orphan objects* in the system once our protocol has finished recovery. The state of the sender of any message, processed by its receiver, should reflect the fact that the message has been already sent.

Alvizi et al. also prove that a receiver side pessimistic protocol that writes to stable storage never produces orphans. However, this proof deals only with the availability of the meta-data of messages within the system and not with the availability of the messages themselves. This is not sufficient for a practical framework that involves checkpoints and garbage collection. We extend the basic idea to prove the correctness of our object based sender side pessimistic message logging protocol that uses periodic checkpoints. The proof deals only with the part of the protocol described till now, that is the basic restart protocol in the face of single failures only. Each object is modeled as a state machine in which processed messages are the only inputs and sent messages are the only outputs. An object transitions from one state to another only as the result of processing a received message. Messages sent to other objects are the only output produced during these transitions.

We define a system $\mathcal{N}$, consisting of n objects running on p processors. The set of objects is $\mathcal{O}$ with the j^{th} object being represented by o_j. The set of processors is $\mathcal{P}$ with the i^{th} processor denoted as p_i. The objects on a processor p_i are represented by the set $O(p_i)$ and the processor of an object o_j is returned by the function $P(o_j)$. As mentioned earlier, objects interact only through messages and messages are the only non-deterministic events affecting the state of objects. Processing message m is represented by the event $process(m)$. These events are ordered by the irreflexive partial order *happens before* $\rightarrow$ that represents potential causality [37]. $process(m_1) \rightarrow process(m_2)$ means that processing m_1 might cause the processing of m_2 and therefore m_1 must always be processed before m_2. The meta-data for a message m, defined as the tuple $< sender, receiver, SN, TN >$ is represented as $|m|$.

Theorem 1 *There is enough information, including meta-data of messages and checkpoints, to avoid orphans at the end of recovery from a single processor failure.*

We define a subset of $\mathcal{O}$, for each message m processed during an execution. This subset $Depend(m)$ is the set of all objects whose state was affected by the processing of m. m *happens before* some message processed by the members of $Depend(m)$. We define a helper boolean function $processed(m)$ that returns true if the event $process(m)$ has occurred. $Depend(m)$ can be formally represented as:

$$\left\{ o \in \mathcal{O} \;\middle|\; \begin{array}{l} ((o = m.reciver) \wedge processed(m)) \\ \vee \left(\exists m' : \begin{array}{l} ((process(m) \rightarrow process(m')) \\ \wedge (o = m'.receiver) \wedge processed(m')) \end{array} \right) \end{array} \right\}$$

We define another set $Log(m)$, which is a subset of $\mathcal{P}$, as the set of proces-

sors that contain information that can prevent m from becoming an orphan. This includes any processor that has a copy of the meta-data of message m or the state of $m.receiver$ once it has processed m. If $m.receiver$ and $m.sender$ objects are on different processors, then according to the remote mode of the protocol $P(m.sender)$ contains a copy of $|m|$ before m is processed. Once $processed(m)$ is true, $Log(m)$ also includes $P(receiver(m))$. If $m.receiver$ checkpoints after processing m, $Log(m)$ includes the buddy of $P(m.receiver)$ (written as $B(P(m.receiver))$ since the checkpoint of $m.receiver$ is stored there. However, after the checkpoint $Log(m)$ no longer contains $P(m.sender)$ since m's entry in $m.sender$'s message log would have been garbage collected after $m.receiver$'s checkpoint. If $m.sender$ and $m.receiver$ are on the same processor then, $|m|$ is saved in the MDTable of $B(P(m.receiver))$ before m is processed. So, $Log(m)$ includes $B(P(m.receiver))$ along with $P(m.receiver)$. Even if $m.receiver$ checkpoints and the entry for m in $B(P(m.receiver))$'s MDTable is deleted, $Log(m)$ still include $B(P(m.receiver))$ as the checkpoint of $m.receiver$ is stored there before garbage collection starts. So, $Log(m)$ once $processed(m)$ is true, has the following values:

$$\begin{cases} \{P(m.receiver), B(P(m.receiver))\} & P(m.sender) = P(m.receiver) \\ \{P(m.sender), P(m.receiver)\} & \begin{cases} P(m.sender) \neq P(m.receiver) \\ \wedge \\ P(m.sender) \text{ has no checkpoints since } process(m) \end{cases} \\ \{P(m.receiver), B(P(m.receiver))\} & \begin{cases} P(m.sender) \neq P(m.receiver) \\ \wedge \\ P(m.sender) \text{ has a checkpoint after } process(m) \end{cases} \end{cases}$$

Now, let us see what happens when p_c fails. An object o on some other processor will become an orphan if $P(o)$ does not fail but the state of o depends

on a message m whose meta-data has been lost. The meta-data of message m can be lost if $Log(m)$ contains only p_c. So the condition for o to be an orphan can be expressed formally as:

$$orphan(o) = \left(\begin{array}{l} P(o) \in \mathcal{P} - p_c \\ \wedge \exists m : ((o \in Depend(m)) \wedge (Log(m) - p_c = \emptyset)) \end{array} \right)$$

A processor is never it's own buddy in our protocol:

$\forall p \in \mathcal{P} : B(p) \neq p$.

Therefore $Log(m)$ is never a set with a single processor, that is:

$\forall m : |Log(m)| > 1$

So, $Log(m) - p_c$ is never an empty set:

$\forall m : Log(m) - p_c \neq \emptyset$

This means that $orphan(o)$ is never true for our system as long as a single processor has crashed:

$$orphan(o) = \left(\begin{array}{l} P(o) \in \mathcal{P} - p_c \\ \wedge \exists m : ((o \in Depend(m)) \wedge false) \end{array} \right) = false$$

Thus, our system has enough information in message meta-datas and checkpoints to avoid orphans at the end of recovery from a single processor failure. This concludes the proof of Theorem 1.

Theorem 2 *All messages, necessary to avoid orphans after recovery from a single processor failure, are available during recovery.*

Now, we aim to prove that all the messages necessary to avoid orphans and not just their meta-data are available during recovery from a single processor failure. The state of an object o_j is represented by $S(o_j)$. The state of a processor p_i at any point is given by the state of all the objects in $O(p_i)$ at

that point. So, $S(p_i) = \{S(o) : o \in O(p_i)\}$. The state of the whole system at some point is specified by the state of all the objects at that time : $S(\mathcal{N}) = \{S(o) : o \in (O)\}$. Each execution of the system is represented by a *run*. A run is a sequence of global states. Each state transition is caused by a single object processing a single message.

A run is punctuated by processors taking checkpoints. The k^{th} checkpoint of object o_j is represented by $S_k(o_j)$. The k^{th} checkpoint of processor p_i is given by $S_k(p_i) = \{S_k(o) : o \in O(p_i)\}$. $S_k(p_i)$ is a state that actually occurs as part of a global state during a run since all the objects on a processor checkpoint at the same time and the checkpoint of a processor can not interrupt the processing of a message. The same can not be said for $S_k((N)) = \{S_k(o) : o \in \mathcal{O}\}$, since the checkpoints of different processors are asynchronous.

$S(o_j) \rightsquigarrow S'(o_j)$ represents a sequence of *process* events that changes the state of object o_j from S to S'. The set of messages processed by o_j during this sequence is represented by $M(S(o_j), S'(o_j))$.

Now we look at what happens after a crash. Let the crashed processor be p_c. Let its state just before the crash be given by $S_c(p_c)$ and its checkpoint just before the crash be $S_k(p_c)$. At the beginning of recovery all objects $o_j \in O(p_c)$ would have been rolled back to $S_k(o_j)$. At this point, the global state is inconsistent. Our recovery protocol aims to remove this inconsistency by eliminating all potential orphans. An object o is a *potential orphan* if it exists on another processor and it's state depends on a message that has not been processed in the system's current state. It must be noted that this definition of potential orphans is subtly different from the previous definition of orphans because it deals with messages themselves and not just their meta-data. So, formally the definition of a potential orphan is if at the beginning of recovery:

$$potential\ orphan(o) = \left(\begin{array}{l} P(o) \in \mathcal{P} - p_c \\ \wedge \exists m : ((o \in Depend(m)) \wedge \neg processed(m)) \end{array} \right)$$

The only messages which have not been processed in the current global state but have affected the state of objects on other processors, are those whose processing events were lost due to p_c's crash and subsequent rollback to the previous checkpoint. The set of such messages (let's call it $MReq$) is given by: $MReq = \bigcup_{\forall o \in O(p_c)} M(S_k(o), S_c(o))$

In order to avoid orphans, we need to prove that all these messages in $MReq$ will be available during recovery. Theorem 1 has already proven that the meta-data for all messages in $MReq$ are present in the system.

Lemma 1 *Every message $m \in MReq$, such that $P(m.sender) \neq p_c$ will be available during recovery.*

A message m whose sender is not on the crashed processor, must have used the remote mode of our message logging protocol. So, $m.sender$ would have stored m in its message log along with $|m|$ before m was processed. Moreover, all messages in $MReq$ were processed by their receivers on p_c after the checkpoint $S_k(p_c)$ had been taken. This means that m's entry in $m.sender$'s message log would not have been garbage collected. So, $m.sender$ can resend m during recovery. Therefore, every message $m \in MReq$, such that $P(m.sender) \neq p_c$ is available during recovery.

Lemma 2 *Every message $m \in MReq$, such that $P(m.sender) = p_c$ will be available during recovery.*

A message $m \in Mreq$ between two objects on p_c would have used the local mode of our message logging protocol (called local messages for brevity). The

sender $m.sender$ does not keep a copy of m anywhere once m has been processed by $m.receiver$. This means that most locals messages in $Mreq$ would have to be regenerated. So, we will try to prove that all the local messages with missing logs will be regenerated during the recovery of p_c.

Let us consider the sequence of process events that would have happened on processor p_c between the checkpoint S_k and the crash S_c. We can write this sequence of events that happened before the crash as $S_k(p_c) \rightsquigarrow S_c(p_c)$.

Let us consider the first local message in the sequence $S_k(p_c) \rightsquigarrow S_c(p_c)$ (call this message m_1). $m_1.sender$ could have sent this message either before or after the checkpoint S_k. First, we look at the case when $m_1.sender$ sent m_1 before the checkpoint. During recovery, $m_1.sender$'s checkpoint would contain m_1 in its message log. This is because the checkpoint of an object includes any messages in it message log that were sent to other objects on the same processor and such a message is removed from the message log only after it has been processed. Since m_1 had not been processed by $m_1.receiver$ by the time the checkpoint S_k happened, m_1 would appear in $m_1.sender$'s message log as part of $m_1.sender$'s checkpoint. Therefore, $m_1.sender$ can simply resend m_1 during recovery. Now for the second case, when m_1 was sent after the checkpoint. All the messages processed in the sequence $S_k(p_c) \rightsquigarrow S_c(p_c)$ before $process(m_1)$ have their senders on some processor other than p_c (by definition of m_1). During recovery, by Lemma 1 all these messages processed before m_1 will be resent and by Theorem 1 their meta-data will also be available. This means that all these messages before m_1 can be processed in the correct sequence by their receivers. This is sufficient to assure that during recovery $m_1.sender$ will pass through the same state that caused it to send m_1 the first time. This means that m_1 would be sent again in this case as well.

Now, let us assume that at least the first i locals messages (m_1 to m_i) in the

sequence $S_k(p_c) \rightsquigarrow S_c(p_c)$ have been re-generated during recovery. Moreover, by Lemma 1 all messages with senders on processors other than p_c can be resent during recovery. Combining these facts with Theorem 1's assertion that the meta-data for all messages needed for recovery are present, we can conclude that all the messages in the sequence $S_k(p_c) \rightsquigarrow S_c(p_c)$ before the $i + 1^{th}$ local message (m_{i+1}) can be processed. This means that during recovery $m_{i+1}.sender$ will process all those messages that it had processed before sending m_{i+1} prior to the crash. Moreover, it will process all those messages in the same sequence during recovery as it did before the crash. This means that $m_{i+1}.sender$ would pass through the same state that caused it to send m_{i+1} the first time.

Thus, if at least the first i local messages have been regenerated the $i + 1^{th}$ local message will also be regenerated. Therefore, by mathematical induction all local messages will be regenerated. This proves Lemma 2.

Lemma 1 and Lemma 2 taken together prove Theorem 2. Theorem 1 and Theorem 2 taken together prove that after a single processor failure, our message logging protocol can assure that there are no orphans at the end of recovery.

Chapter 5

Multiple Simultaneous Failures

The protocol discussed in Chapter 4 works for all single processor failure cases. As long as all processors have checkpointed at least once between two processors crashing, the application can recover from the crashes and continue execution. However, in systems with multi-processor nodes all the processors in a node would probably go down at the same time. Therefore, we study the reasons for this restriction and extend the protocol to allow it to deal with most multiple failures. However, the assumption that a processor and its buddy do not go down at the same time is still maintained.

5.1 Challenges and Solutions

This section describes the exact challenges faced by the current message logging protocol when trying to recover from multiple processor failures within a short time. It also discusses how the current protocol can be modified to meet these challenges. There are two major sources of problems:

- Missing message logs
- Missing message meta-data

We discuss the exact sources of these two challenges and their solutions in the following subsections.

5.1.1 Missing Message Logs

Figures 5.1 and 5.2 illustrate a problem faced by the simple message logging protocol when two processors crash close to each other. Figure 5.1 shows the initial state of our example. Object γ on processor H sends a message m to object δ on processor I. According to the remote mode of the message logging protocol, γ saves a log of m, including m's meta-data in its message log. For the sake of clarity, we skip the protocol messages in this figure. To simplify our example we also assume that processor H is not a buddy of I and vice-versa. Processor H takes a checkpoint after m has been sent. At the checkpoint object γ's state reflects the fact that it has already sent message m. However, processor I checkpoints before m has been processed. This is shown in Figure 5.1 by the fact that when processor I checkpoints object δ has $TNProcessed = 184$, while the TN of message m is 187. Some time after the checkpoint, processor H crashes.

In stage 2, shown in Figure 5.2(a) processor H starts recovery by fetching its checkpoint from its buddy. Processor H recreates object γ from this checkpoint. However, in our message logging protocol the logs of messages sent to objects on other processors are not part of the checkpoint state of the sender. Therefore, when object γ is recreated, its message logs do not contain a log of message m. Moreover, according to its current state γ has already sent message m and will not regenerate it. A little later, before processor I has had the chance to checkpoint again, processor I crashes.

In stage 3 in Figure 5.2(b) processor I is now restarted using its previous checkpoint. Object δ is recreated and starts its recovery. At this point, it has a $TNProcessed = 184$. At some point during recovery, object δ will need to process message m with a TN equal to 187. However, as explained earlier,

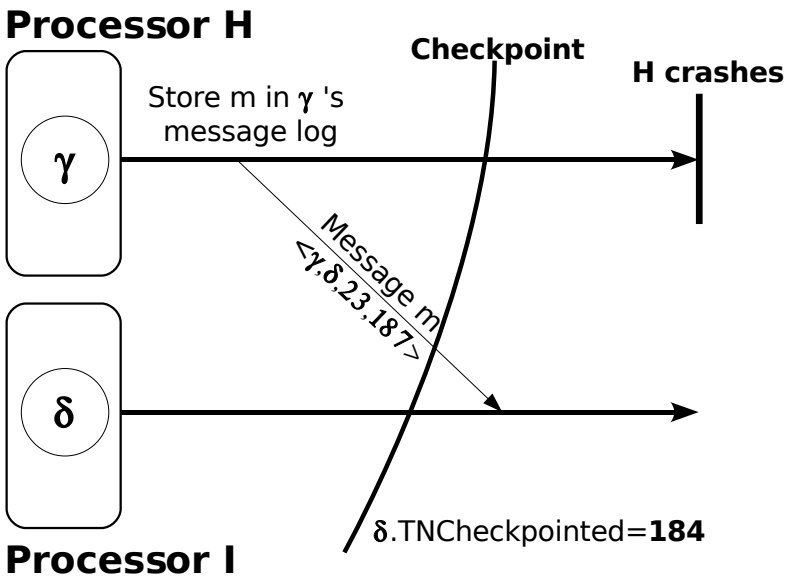
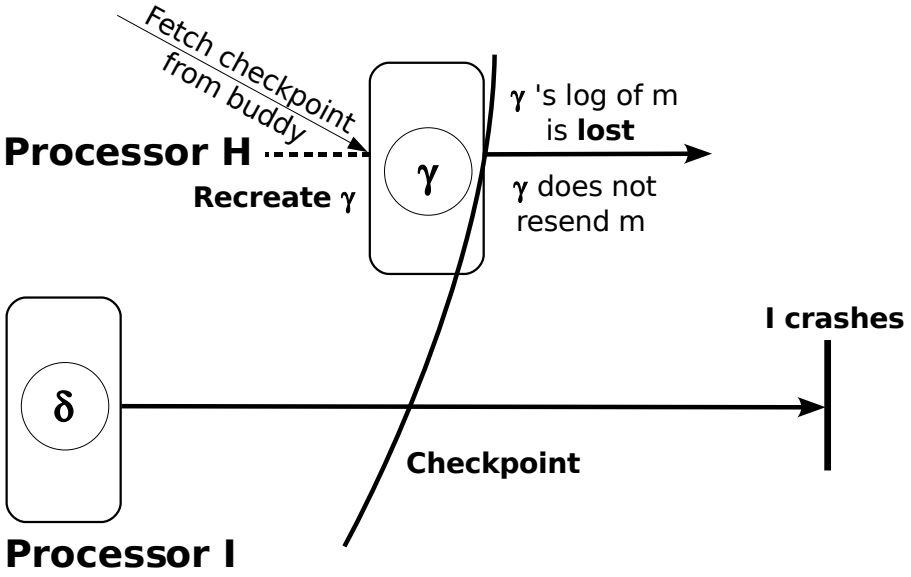


Figure 5.1: Object γ on processor H sends a message to δ on processor I. After that processor H takes a checkpoint and then crashes. This is Stage 1 of the problem of lost message logs which occurs when two processors crash within a short time of each other. Figure 5.2 contains the remaining two stages.

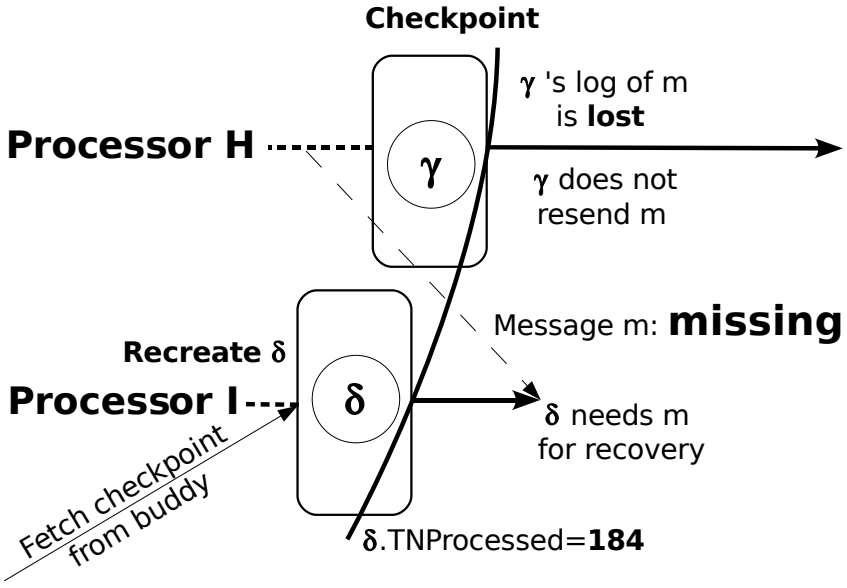
object γ will not resend message m. Therefore, during recovery object δ is never going to get message m from object γ. So the recovery of object δ on processor I gets stuck.

The solution involves forcing object γ to resend message m. One way to do this would be to roll processor H back to an even earlier checkpoint. This would force object γ to re-execute and re-generate message m. However, this would fail to meet one of the primary aims of our fault tolerance protocol that the crash of one processor should not cause another to be rolled back. Moreover, it would require each processor to store multiple checkpoints increasing the overhead of our protocol. Therefore, rolling back processor H even further can not be a solution to our problem.

We solve the problem by making the entire message log of object γ and its TNTable a part of its checkpoint state. Earlier, we only saved those entries in



(a) Stage 2: Processor H is recovering. Processor I crashes



(b) Stage 3: Processor I's recovery gets stuck.

Figure 5.2: Illustrates the problem of lost message logs faced by the simple message logging protocol when two processors crash within a short time of each other

the message log that contained messages to other objects on the same processor as γ during its checkpoint. Now, we save all entries in its message logs to all objects whether on processor H or not. When object δ begins recovery,

object γ resends all messages in its message log with TN higher than 184 (δ's $TNProcessed$ at the checkpoint). Since message m has a TN of 187 it gets resent and reprocessed by object δ. It should noted that we are **not** suggesting that all message logs from the very beginning of the program are a part of γ's checkpoint state. Garbage collection still continues, with messages in message logs being deleted whenever their receivers checkpoint.

5.1.2 Missing Message Meta-data

If two processors crash very close to each other, the meta-data of some messages can be lost during recovery in our current message logging protocol. This means that during recovery a message can get a TN different from what it got before the crashes. We illustrate how this can happen with an example shown in Figures 5.3 and 5.4. In Stage 1 of the example, shown in Figure 5.3, the first thing to occur is the checkpoint of processors H and I. Objects γ and ϵ save their states as a part of processor H's checkpoint. Object δ is a part of the checkpoint of processor I. At the time of the checkpoint, the highest ticket handed out by object δ is 187. After the checkpoint, objects ϵ and γ send messages m and n respectively to object δ. Object ϵ sends message m as a result of receiving message a from an object on some processor other than H or I. Similarly object γ sends message n as a result of receiving message b from yet another processor. It should be noted that there is no predefined ordering between when messages a and b are processed. In this case as message a happens to arrive first, object ϵ processes it before object γ processes message b. As a result, object ϵ sends its ticket request for message m before object γ sends one for message n. Object δ assigns message m a TN of 188 by incrementing $TNCount$. It gives message n a TN of 189. When the tickets are received by objects ϵ and γ, the tickets are stored in the message logs and the messages are sent to object δ. Object δ

processes the messages in the increasing order of TN: first m and then n. At the end of this, processor H crashes.

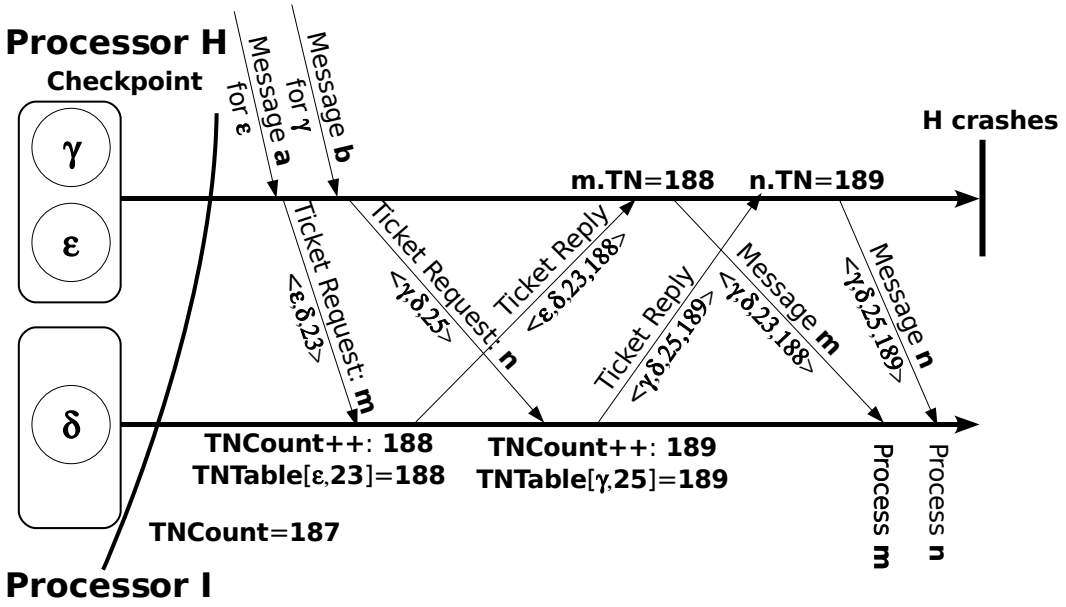
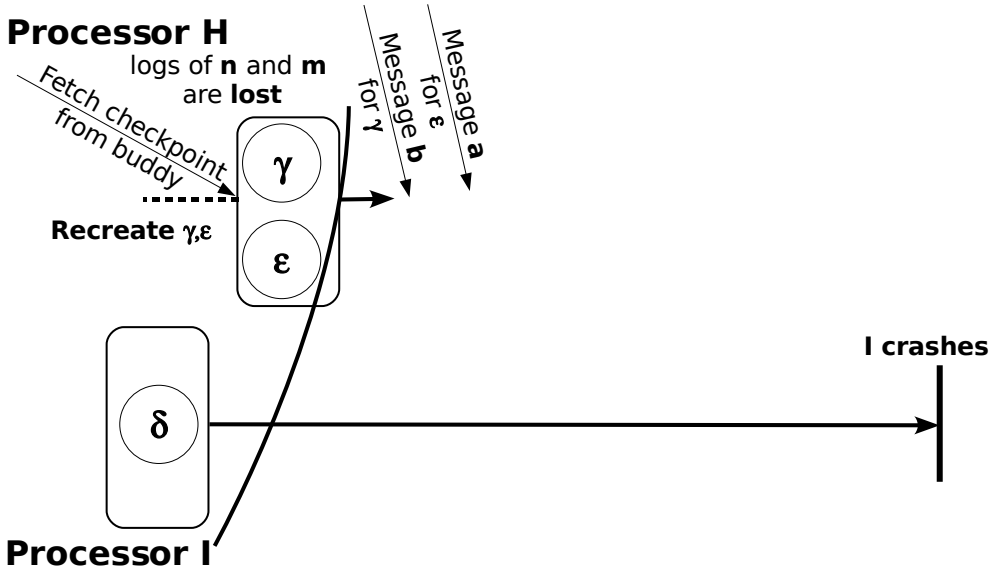


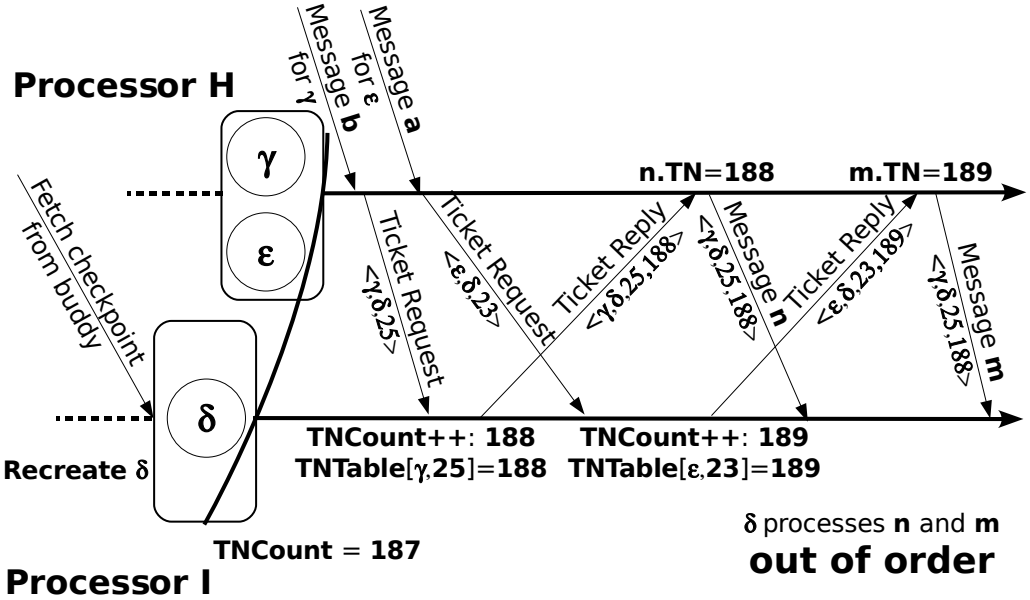
Figure 5.3: Stage 1 of the problem of lost message data which occurs when two processors crash within a short time of each other. Figure 5.4 contains the remaining two stages

Stage 2 in Figure 5.4(a) has processor H starting its recovery. It fetches its checkpoint from it buddy. It recreates objects ϵ and γ and restarts their execution. At this point objects ϵ and γ do not contain the log of messages m and n respectively. The logs are not part of their checkpoint state since the messages were sent after the checkpoint as shown in Figure 5.3. Objects on other processors resend messages to objects ϵ and γ. Messages a and b are also resent by their senders. Unlike the situation in Figure 5.3, during recovery message b arrives before message a. However, before messages b or a can be processed, processor I crashes.

Stage 3 in Figure 5.4(b) shows processor I recovering from the crash. It gets its checkpoint from its buddy and recreates object δ. At this point, its TNTable does not contain any entries for TNs assigned to messages after its



(a) Stage 2: Processor H is recovering. Processor I crashes



(b) Stage 3: Processor H and I both continue their recovery

Figure 5.4: Illustrates the problem of lost meta-data for regenerated messages faced by the simple message logging protocol when two processors crash close to each other

checkpoint (namely messages m and n). Moreover, the $TNCount$ for object δ is 187 at the start of recovery. Object δ sends out a request to resend messages to it. However, messages m and n will not be resent because of this request

since their logs were lost when processor H crashed and forced objects ϵ and γ to restart from their previous checkpoint. Instead, messages m and n will be resent because objects ϵ and γ re-process messages a and b respectively during their own recovery. However, this time message b for γ arrives before message a for ϵ. This happens because messages a and b are sent from different processors to different objects on processor H. When message b is processed, object γ sends out a ticket request for message n to δ, while object ϵ processing a leads to a ticket request for message m. When object δ receives the ticket request for message n, it has no idea that it had ever allotted a TN to this message. So object δ treats message n as a new message. It increments $TNCount$ and gives n a TN of 188. Similarly, object δ assigns message m a TN of 189. Object δ then proceeds to process these messages in the increasing orders of their TNs. As a result, object δ ends up processing message n before m. However, in Figure 5.3 δ had processed message m first and then n. This invalidates the primary aim of the recovery protocol that an object should process the same messages in the same sequence after a crash as before it.

This problem can be solved by borrowing an idea from the local mode of operation of the message logging protocol. Figure 5.5 shows how the modified version of the remote mode works. In this example object γ on processor H sends message n to to object δ on processor I. The first step is the same as earlier, object γ assigns a SN (24) to the message n and saves a log of n. However, the sender object γ no longer sends out a ticket request to the receiver object δ but sends the message itself along with the sender id, receiver id and SN tuple. The receiver δ, on processor I, calculates a new TN (189 in this case) and stores an entry for it in the $TNTable$. It then sends the meta-data of this message n, consisting of the sender id, receiver id, SN and TN tuple to the buddy of processor I, namely processor J. Processor J stores this meta-data

in its MDTable and sends an acknowledgment for message n back to object δ on processor I. Once object δ receivers this acknowledgment, it can process message n in increasing order of TN.

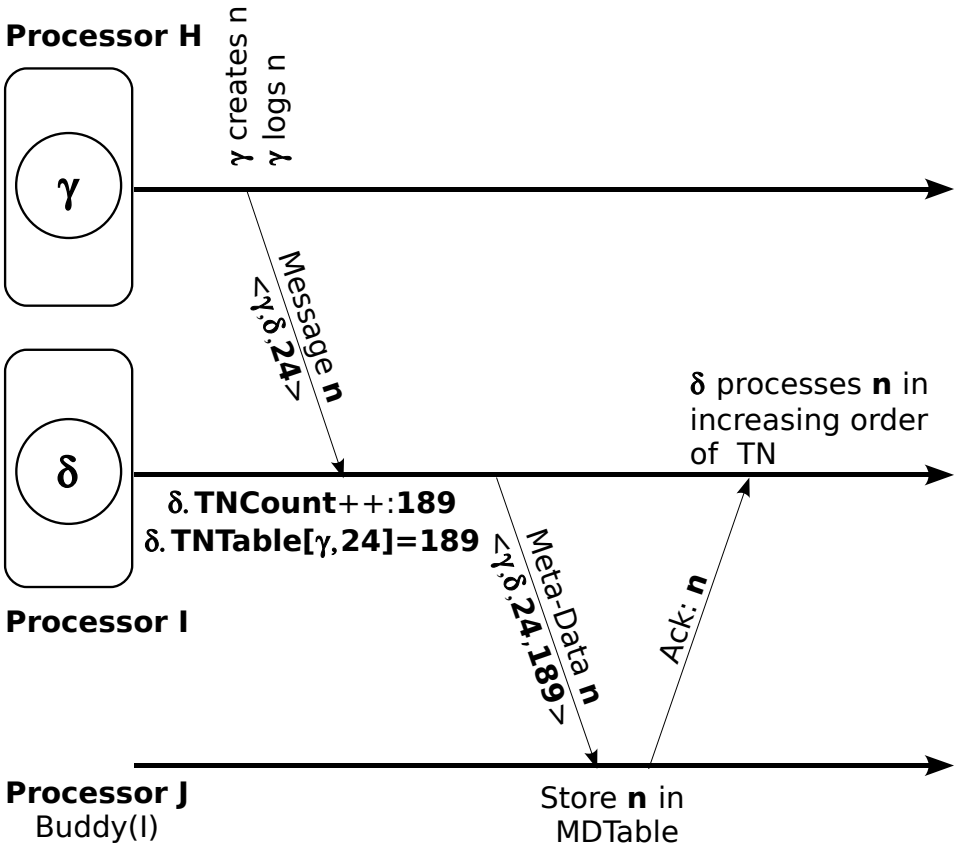


Figure 5.5: The modified version of the remote protocol designed to deal with multiple failures

If processor I were to crash, object δ would be restarted from a previous checkpoint stored on processor J. Processor J would also send the message meta-data stored in $MDTable$ along with the checkpoint. The meta-data for different messages fetched from the $MDTable$ would be stored as entries in object δ's $TNTable$. Object δ would then ask all objects to resend messages to it. Object γ would re-send message n from its log. However, this message would not contain a TN. Instead, object δ would look up the message n's TN in its TNTable. It is bound to find an entry in the TNTable as long as the message

is an old one that was processed before the crash. Thus, we solve the problem of missing meta-data in the face of multiple failures by making sure that the meta-data of all messages processed by an object are stored on the buddy of its processor. This reduces the restarted object's dependency for meta-data of processed messages on multiple objects on multiple processors. It must be noted that we are saving just the meta-data of messages on the buddy and not the messages. The messages would still be logged and resent from (in the case of single failure) or regenerated by (in the case of simultaneous failure) the sender objects.

We implement the improvements for tolerating multiple faults, but let the user turn it off to avoid the overhead of checkpointing message logs, if he thinks that the chances of simultaneous failures are low.

The only case in which our solution might fail occurs when processor I crashes just after its buddy processor J has crashed and restarted. As processor J no longer has processor I's MDTable, the objects on I cannot successfully recover. The probability of such a pair of unrecoverable crashes happening can be reduced by having processor I checkpoint as soon as processor J restarts. Once processor I has checkpointed, the objects on I do not need the meta-data stored in the MDTable on processor J that were lost when J crashed. This shortens the length of the time window during which a crash might cause an unrecoverable error. This situation arises because unlike [13, 14] we do not use an idealized stable storage.

5.2 Reliability Improvement Analysis

We present a calculation based on a simple model to prove that our protocol increases the reliability of a system, in spite of not being foolproof. Let the

number of processors be p. Let the failure rate of a single processor be λ. Let λ be the same for all p processors. Let the run time of the application without a fault tolerance protocol be R units. The probability of a particular processor crashing during the runtime of the application can be approximated by λR. The probability of a particular processor not failing through out the runtime of an application is then $1 - \lambda R$. The probability of none of the processors failing during the application's execution is given by $(1 - \lambda R)^p$. Therefore, the probability that the application will fail $= 1 - (1 - \lambda R)^p$ (1).

Now, we consider the case when the application uses our fault tolerance protocol. Let the run time of the application in this case be R' units, where $R' > R$, to account for the unavoidable performance cost of fault tolerance. The probability of any particular processor crashing during the runtime of the application is $\lambda R'$. The application would now face an unrecoverable error only if the buddy of the crashed processor crashes before taking a checkpoint. In our protocol, processors are free to take their checkpoints at any point of time. However, practical considerations such as growing message log size normally place an upper bound on the time difference between two consecutive checkpoints taken by a processor. This upper bound also determines the amount of computation that has to be redone due to a crash. We shall see that it is also a factor in the reliability of the system. In order to simplify the analysis, we assume that the upper bound is t units of time for all processors.

It is also assumed that the probability of failure of a processor doesn't change after its buddy crashes. Choosing a buddy, such that the co-relation of failure between a processor and its buddy is low, validates this assumption to a large extent. Since each processor is assumed to take a checkpoint at least once every t units, the probability of an unrecoverable error, given that a processor has crashed, is λt. So the probability of a particular processor causing an unrecover-

able fault is $(\lambda t)(\lambda R') = \lambda^2 t R'$. A processor runs successfully with a probability of $1 - \lambda^2 t R'$. The probability of a successful run, with no processor suffering an unrecoverable crash is given by $(1 - \lambda^2 t R')^p$. Therefore the probability of an unrecoverable fault during the application run is $1 - (1 - \lambda^2 t R')^p$ (2).

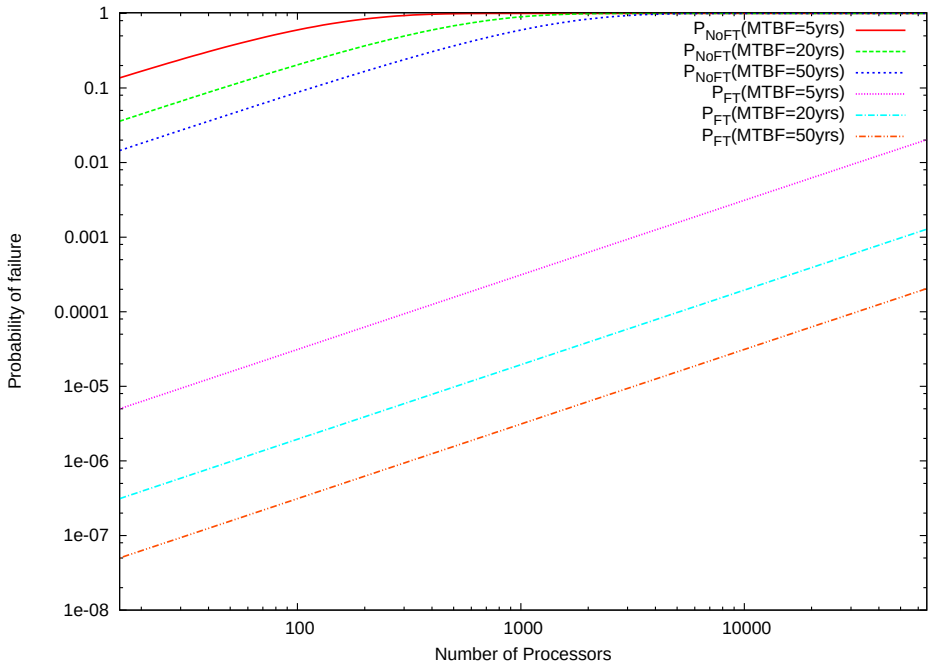


Figure 5.6: The probability of failure for runs without and with our fault tolerance protocol for different values of Mean time between failures(MTBF): 5,20,50 years. The probability of failure is plotted against the numbers of processors. R (runtime without fault tolerance)=400. R'(runtime with fault tolerance)=1200.t(time between checkpoints)=.5$hours$.

In order to bring out the huge difference between the expressions in (1) and (2), we evaluate them for a range of plausible system parameters. Figure 5.6 shows the probability of failure for a particular runtime duration without(P_{NoFt}) and with (P_{FT}) our fault tolerance protocol. The probability of failure is calculated for an execution time of 400 hours without fault tolerance and 1200 hours with fault tolerance. The same execution time is used for all numbers of processors. We choose a 3 time increase in run-time as a worst case scenario since our protocol increases short message latency to 3 times. We plot the prob-

ability of failure for 3 different values of mean time between failure(MTBF) for each processor: 5 years, 20 years and 50 years. For our fault tolerance protocol, we assume a processor checkpoints at least once every half an hour, therefore $t = .5$ *hours*. Figure 5.6 clearly demonstrates that our protocol drastically reduces the probability of failure, even on large machines made up of unreliable components. Without any fault tolerance support, the chance of failure rises sharply with increasing numbers of processors. It becomes impossible to get successful runs with probability of failure reaching 1 when running on 1000 or more processors. Our protocol, on the other hand even with very unreliable components (MTBF=5years) still manages to run 99% of the time even on massive machines with tens of thousands of processors. With moderately reliable components our protocol has a very low property of failure on large machines.

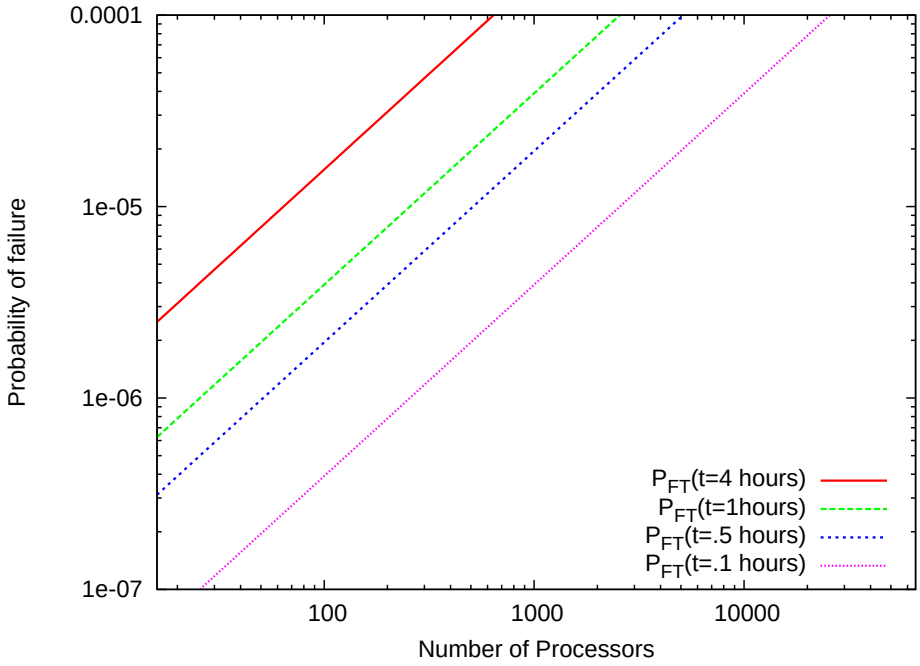


Figure 5.7: Probability of failure of runs with our fault tolerance protocol, on different numbers of processors, for different values of t(the maximum time between two checkpoints). The execution time R' is 1200 *hours* and the MTBF for individual processors is 20 *years*.

Figure 5.7 shows the variation in the probability of failure of our fault tol-

erance protocol with different periods of checkpointing. The higher the time between checkpoints, the higher is the probability of an unrecoverable occurring during a run. This makes sense intuitively since a higher time between checkpoints, means that a crashed processor's buddy has a longer time during which if it fails, the application will suffer an unrecoverable error. So when a processor crashes it is important for its buddy to checkpoint and try to reduce the window of time during which a crash can lead to an unrecoverable error. If one does not do that while using our protocol, then the checkpoint period becomes a factor in the reliability of the protocol itself. This complicates the traditional tradeoff in message logging protocols between checkpoint frequency and the size of message logs.

5.3 Proof of Correctness

Theorem 3 *There is enough information, including meta-data of messages and checkpoints, to avoid orphans at the end of recovery from multiple simultaneous processor failures as long as a processor and its buddy do not crash at the same time.*

We use the definition of $Depend(m)$ from Chapter 4.4 as the set of all objects whose state was affected by the processing of message m. Just for the sake of clarity we present the formal definition of $Depend(m)$ again as:

$$\left\{ o \in \mathcal{O} \;\middle|\; \begin{array}{l} ((o = m.reciver) \wedge processed(m)) \\ \vee \left(\exists m' : \begin{array}{l} ((process(m) \rightarrow process(m')) \\ \wedge (o = m'.receiver) \wedge processed(m')) \end{array} \right) \end{array} \right\}$$

The definition of $Log(m)$, as the set of processors that contain information that can prevent message m from becoming an orphan, is also reused.

However, with the changes to the remote mode of the message logging protocol the value of $Log(m)$ changes from that in Section 4.4. If $m.receiver$ and $m.sender$ are on different processors, the meta-data of m is stored in the $MDTable$ of the buddy of $P(m.receiver)$ instead of $P(m.sender)$. Even after a checkpoint of $P(m.receiver)$, when $|m|$ gets garbage collected from the $MDTable$ on $B(P(m.receiver))$ the checkpoint of $m.receiver$ itself is stored on $B(P(m.receiver))$. So, $Log(m)$ always includes $B(P(m.receiver))$ for the remote mode of the protocol, the same as the local mode. As a result $Log(m)$ now has the same value for all messages m as long as $processed(m)$ is true. Therefore, the value of $Log(m)$ can be formally written down as:

$Log(m) = \{P(m.receiver), B(P(m.receiver))\}$

Now, a set of processors $\mathcal{C} \subset \mathcal{P}$ crash. It should be noted that our protocol assumes that this does not include a processor and its buddy. Therefore, $\mathcal{C}$ is a set of the following form:

$\mathcal{C} = \{p_c : p_c \in \mathcal{P} \wedge B(p_c) \notin \mathcal{C}\}$

An object o on some other processor will become an orphan if $P(o)$ does not crash but the state of o depends on a message m whose meta-data has been lost. The meta-data of a message can be lost if $Log(m)$ is a subset of $\mathcal{C}$. We express the condition for o to be an orphan formally as:

$$orphan(o) = \left(\begin{array}{l} P(o) \in \mathcal{P} - \mathcal{C} \\ \wedge \exists m : ((o \in Depend(m)) \wedge (Log(m) \subseteq \mathcal{C})) \end{array} \right)$$

Since, for every p_c in $\mathcal{C}$ its buddy is not in C: $Log(m) \nsubseteq \mathcal{C}$

Therefore, $orphan(o)$ is never true for our system as long as a processor and its buddy have not crashed together:

$$orphan(o) = \left(\begin{array}{l} P(o) \in \mathcal{P} - \mathcal{C} \\ \wedge \exists m : ((o \in Depend(m)) \wedge false) \end{array} \right) = false$$

Therefore, our protocol has enough information in message meta-datas and checkpoints to avoid orphans at the end of recovery from multiple simultaneous processor failures as long as a processor and its buddy have not failed at the same time. This completes the proof of Theorem 3.

Theorem 4 *All messages, necessary to avoid orphans after recovery from a multiple simultaneous processor failure, are available during recovery as long as a processor and its buddy do not crash.*

The case for the multiple failure case is different from that in Theorem 2 only for the messages sent between objects on the crashed processors. If objects on the crashed processors had not sent any messages to each other, there would be no difference between the two cases. The recovery on the different crashed processors would simply proceed independent of each other. So, we concentrate on messages exchanged between objects on the crashed processors. We will try to prove that all such messages along with local messages between objects on the same processor will be regenerated. The proof will turn out to be quite similar to that of Lemma 2.

For every processor $p_c \in \mathcal{C}$, the state of p_c at the checkpoint just before the crash is given by $S_k(p_c)$ and its state just before the crash is given by $S_c(p_c)$. For all $p_c \in \mathcal{C}$, consider the process events in the sequence $S_k(p_c) \rightsquigarrow S_c(p_c)$. Let us interleave all these process events on the different crashed processors into one sequence such that causality is never violated (if $process(m_a) \rightarrow process(m_b)$, event $process(m_a)$ occurs before event $process(m_b)$ in the interleaved sequence). We are sure that such a sequence exists because, it is possible to have an execution in which we place all objects on the crashed processors (the set given by

$\bigcup_{p_c \in \mathcal{C}} O(p_c)$) on one single processor such that all messages have the same meta-data as before. This is possible because in a virtualized system, placement of objects on processors does not impact the state of the object. Let us represent one such interleaved sequence by $S_k(\mathcal{C}) \rightsquigarrow S_c(\mathcal{C})$. Of course, any process event m in this sequence such that $P(m.sender) \in \mathcal{P} - \mathcal{C}$, will be resent according to our recovery protocol.

Consider the first process event in the sequence $S_k(\mathcal{C}) \rightsquigarrow S_c(\mathcal{C})$, such that the sender of the corresponding message exists on a crashed processor (call this message m_1). There are two possibilities: message m_1 was sent before or after $m.sender$ checkpointed. If it is the former, m_1 would be a part of the checkpoint state of $m_1.sender$. This is true, irrespective of whether $P(m_1.sender) = P(m_1.receier)$, since the modified version of the protocol saves all entries in $m_1.sender$'s message log as a part of its checkpoint state. Now for the second case, in which m_1 was sent after $m_1.sender$ checkpointed. All the messages processed in the sequence $S_k(\mathcal{C}) \rightsquigarrow S_c(\mathcal{C})$ before $process(m_1)$ have their senders on some processor $p \in \mathcal{P} - \mathcal{C}$. This is true by the definition of m_1. Moreover, one such process event in the sequence $S_k(\mathcal{C}) \rightsquigarrow S_c(\mathcal{C})$ **earlier** than the event $process(m_1)$ is responsible for sending message m_1. This is bound to be true since the interleaved sequence does not violate causality. Since all messages with senders on processor other those in $\mathcal{C}$ will be resent and by Theorem 3 their meta-data will be available, all the messages in the sequence before m_1 will be processed. This means that during recovery $m_1.sender$ is bound to pass through the same state that caused it to send m_1. Therefore m_1 is bound to be available during recovery.

Now, let us assume that at least the first i messages in the sequence $S_k(\mathcal{C}) \rightsquigarrow S_c(\mathcal{C})$ that have their senders on crashed processors have been re-generated during recovery. Moreover, all the messages in the sequence with senders on

processors other than the crashed ones will be resent. Theorem 3 also states that the meta-data of all these messages will be available during recovery. Therefore, at least all messages in the sequence $S_k(\mathcal{C}) \rightsquigarrow S_c(\mathcal{C})$ before the $(i+1)^{th}$ message (m_{i+1}) with a sender on a crashed processor will have been processed. This means that the object $m_{i+1}.sender$ will pass through the same state that led it to sending m_{i+1} in the first place. Therefore, if at least the first i messages with a sender on a crashed processor have been regenerated, the $i + 1^{th}$ will also be regenerated. By mathematical induction, all messages with senders on crashed processors will be regenerated. This along with the fact that all messages with senders on uncrashed processors will be resent means that all messages necessary for an orphan less recovery in the face of multiple simultaneous failures (but not involving any processor and its buddy) will be regenerated.

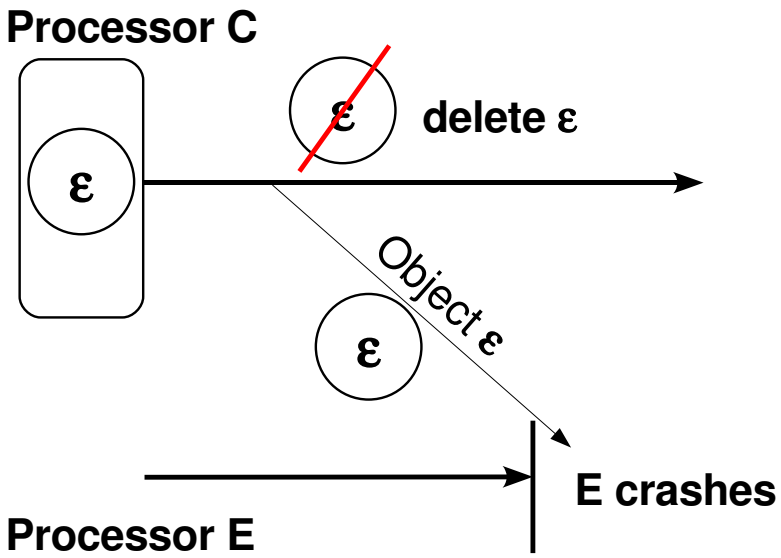
Chapter 6

Fast Restart

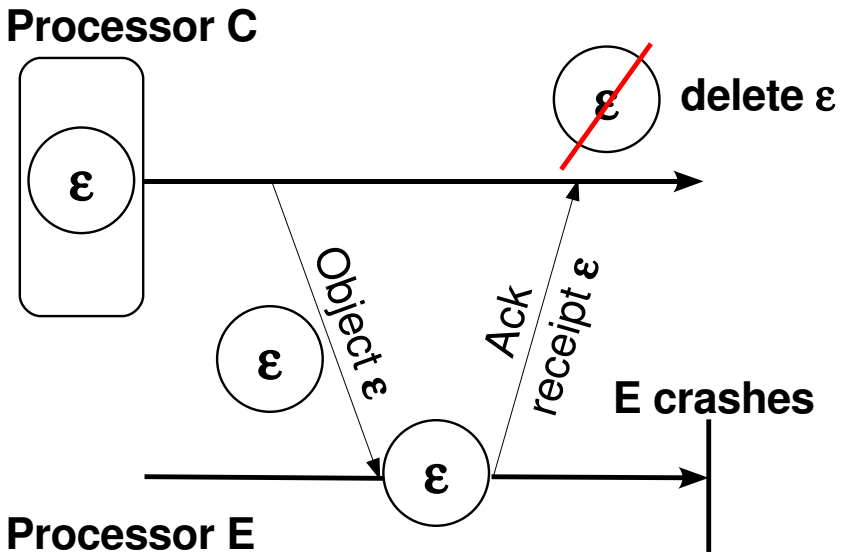
We discuss the fast restart protocol in this chapter. The fast restart protocol makes use of object based virtualization to distribute the work on the recovering processor among other processors. This parallelizes the recovery of the crashed processor, leading to faster recovery. This is expected to be particularly useful for scientific applications which are tightly coupled. While a processor is recovering in a tightly coupled application, other processors start waiting for data from the recovering processor. Our fast restart protocol utilizes these waiting processors to shorten the time wasted in waiting for the recovery to finish.

However, moving objects from one processor to another is a fraught process. Figure 6.1(a) shows one of the problems faced while moving objects from one processor to another. Object ϵ is to be moved from the recovering processor C to processor E. Processor C packs up the state of object ϵ and sends it in a message to processor E. Processor C deletes its copy of object ϵ after sending the message containing ϵ. Now if processor E were to crash before the message got to processor E, neither processor C nor E would have a copy of ϵ. Moreover, during processor E's recovery ϵ would not be created since its buddy would not have object ϵ's checkpoint. One could think of solving this problem, by having processor C not delete its copy of object ϵ until processor E acknowledges that it has received and created object ϵ on itself. However, this solution merely postpones the problem. Figure 6.1(b) shows the situation when processor E crashes after it has sent an acknowledgement to processor C. As a

result, processor C does not contain a copy of object ϵ and moreover processor E's buddy does not contain a checkpoint of object ϵ. Thus object ϵ will not be re-created during processor E's recovery.



(a) Object ϵ lost during migration from processor C to E



(b) Object ϵ lost after migration from processor C to E

Figure 6.1: Examples of problems faced while trying to parallelize restart by moving objects from one processor to another.

We developed a fast restart protocol that migrates objects from the recovering processor to another, while making sure that if another processor were

to crash, all lost objects would be recreated and only one copy of each object would be created. Figure 6.2 shows the messages involved in the fast restart protocol. Object ϵ is to be moved from processor C to processor E. Processors D and F are the buddys of processors C and E respectively. Processor C marks object ϵ as **migrating** to processor E. It also sends a message to this effect to its buddy processor D. On receiving that message processor D marks object ϵ as **migrating** to processor R in ϵ's checkpoint stored on processor D. Processor D then sends an acknowledgment message back to processor C. At this point, processor C packs the state of object ϵ into a message. The message containing object ϵ is sent to the destination processor E and its buddy processor F. Processor C does not yet delete its copy of object ϵ. However, it does stop object ϵ from processing any further messages and buffers any messages for ϵ.

When processor E gets the message containing object ϵ, it stores the message and sends acknowledgments to processors C and D. After processor F gets the copy of object ϵ, it adds ϵ to its copy of processor E's checkpoint. After that processor F sends acknowledgments to processor C, D and E. Once processor C receives the acknowledgments from both processors E and F, it can delete its copy of object ϵ. Processor D waits to get an acknowledgment each from processor E and F and then marks object ϵ as **migrated** in its checkpoint of processor C. Processor E waits for the acknowledgment from processor F before creating object ϵ from the stored message. It then allows object ϵ to start processing messages. At this point object ϵ has successfully migrated from processor C to E. If processor E were to crash now, object ϵ would be re-created on processor E from its checkpoint on processor F.

We now briefly discuss how the fast restart protocol behaves if one of the four involved processors crashes, before the fast restart protocol has finished. During a fast restart if processor C crashes again before its buddy processor D

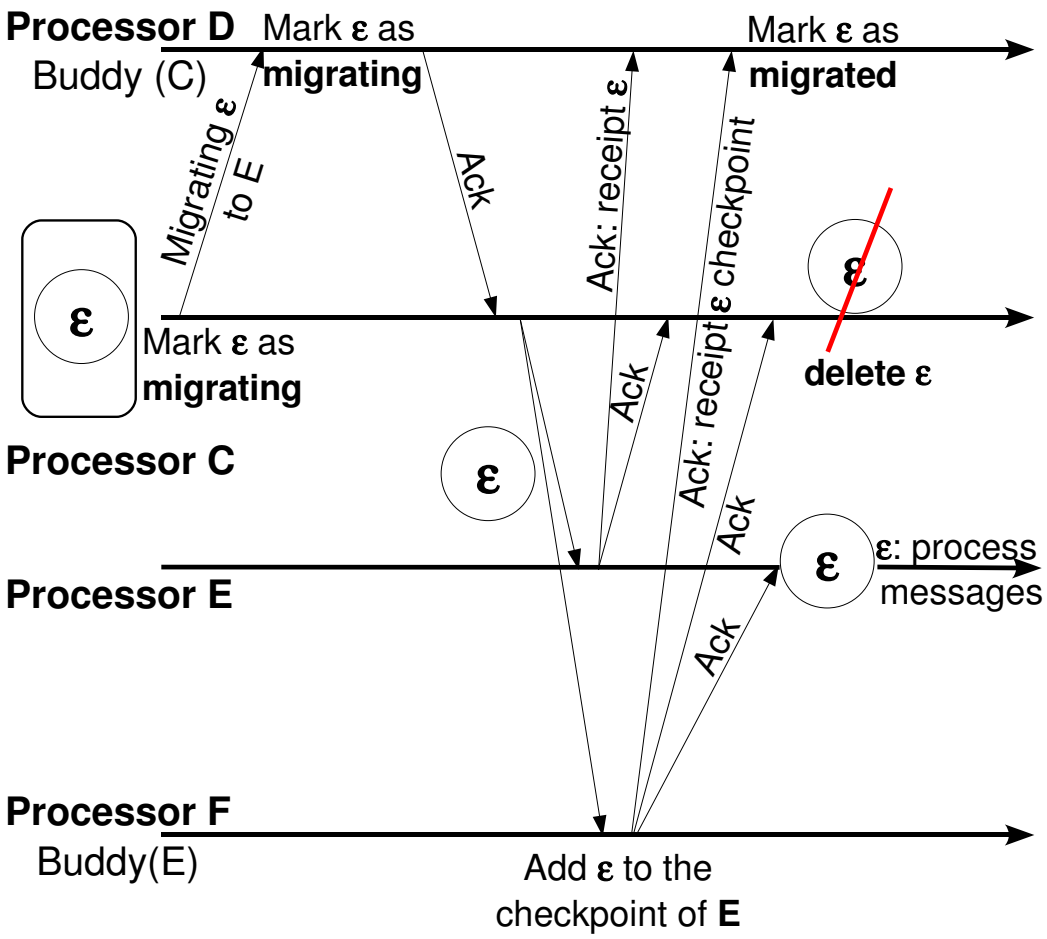


Figure 6.2: Messaging when processor C sends object ϵ to restart on processor E

has received the acknowledgments from processors E and F, D asks if object ϵ and its checkpoint exist on E and F respectively. Processor E stops processing messages for object ϵ after being asked this question. If both processors E and F answer in the positive, processor D does not recreate object ϵ on C and asks E to continue with the execution of messages for ϵ. If not, it recreates object ϵ on processor C and asks processors E and F to throw away object ϵ and its checkpoint respectively. The case in which processor E but not F has received object ϵ and processor E crashes can be resolved by continuing with ϵ's execution on processor C after confirming that processor F does not indeed

have ϵ's copy. If processor E crashes after processor F has got the copy of object ϵ, the object can simply be recreated on processor E from the copy on processor F. In that case processor D does not recreate object ϵ on processor C since it finds that processor F has a copy.

Though the fast restart protocol is more complicated than the basic one, the speed up in recovery gained by dividing the work among multiple processors more than makes up for the additional overhead. The small number of short messages sent during fast restart is overshadowed by the large number of messages being resent as a part of the restart. Fast restart can potentially significantly shorten the recovery time for an application.

6.1 Analysis

We do a rough analysis of our fast restart protocol. We compare the completion time of an application running the fast restart protocol with the same application running a traditional checkpoint/restart protocol.

Let the mean time between failure for the system be m time units.

Let the system checkpoint every c time units (not including the checkpoint period itself).

Let duration of a checkpoint be d.

Let the runtime of the application without any fault tolerance support be t_0.

So time to complete the application with checkpoints: $t_c = t_0 + \frac{t_0}{c}d$.

If there are n faults, the worst case runtime under the checkpoint scheme will be

$t'_c = t_c + n(c + k_c)$ where k_c is the constant overhead for restarting in the checkpoint scheme.

On an average, we expect $n = \frac{t'_c}{m}$ faults during a run, so the execution run time

is given by $t'_c = \dfrac{t_0(1+\dfrac{d}{c})}{1-\dfrac{c+k_c}{m}}$.

t'_c goes rapidly to infinity as m approaches $c + k_c$. In other words as the mean time between failure approaches the worst case recovery time per failure, the application can make very little progress and spends most of the time just recovering from faults. As a result the total execution time blows up.

For the message logging protocol, runtime without faults is
$t_{ml} = r(t_0 + \dfrac{t_0}{c}d)$ where r is the ratio of increase in runtime due to the message logging protocol.
If the number of objects per processor is v and k_{ml} the overhead of fast restart, then the runtime with faults can be calculated to be

$$t'_{ml} = \frac{rt_0(1+\dfrac{d}{c})}{1-\dfrac{\dfrac{c}{v}+k_{ml}}{m}}.$$

The runtime for the message logging protocol goes to infinity rapidly as m approaches $\dfrac{c}{v} + k_{ml}$. In the case of the message logging protocol too, execution time increases sharply once the mean time between failure becomes close to the recovery time. The difference between the two protocols lies in the fact that the recovery time for the message logging protocol is lower than that of the checkpoint protocol in most cases. As long as k_{ml} is not much larger than k_c, $\dfrac{c}{v} + k_{ml}$ is smaller than $c + k_c$ and the message logging protocol can tolerate a higher rate of failure than a checkpoint based protocol.

This shows that our fast recovery protocol can deal with higher rates of failure than the checkpointing protocol. Moreover the performance of the fast

protocol is better than the checkpoint protocol as long as

$$r < \frac{m - (\frac{c}{v} + k_{ml})}{m - (c + k_c)}.$$

This gives us a condition that can be used to decide whether for a given application and machine our protocol will perform better than a traditional checkpoint based protocol.

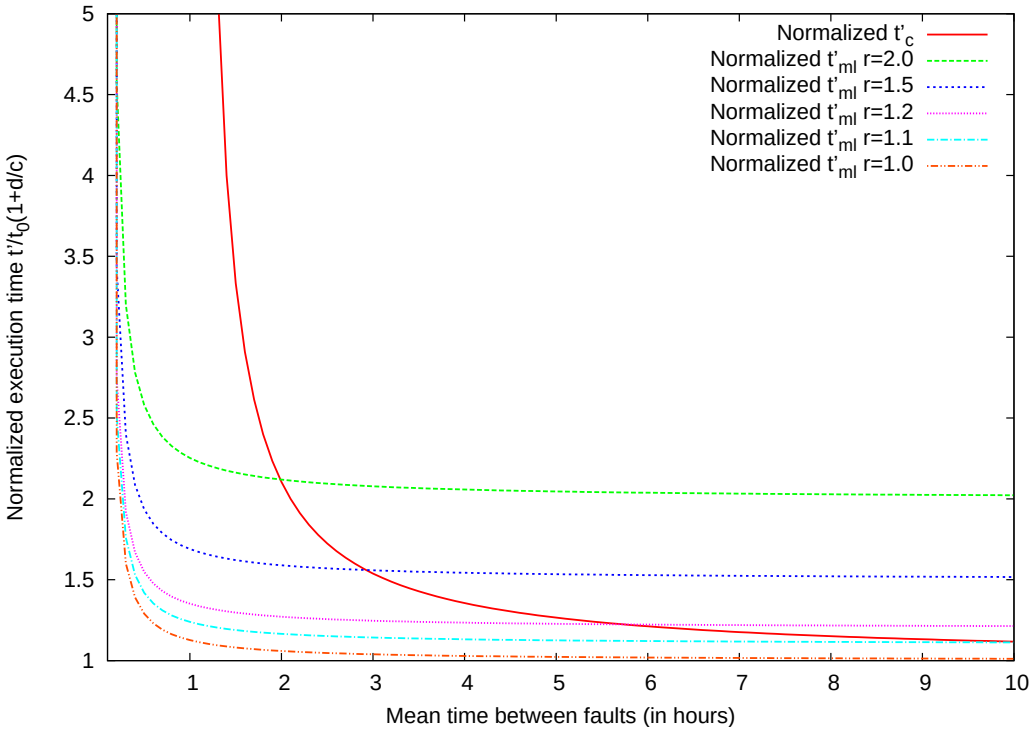


Figure 6.3: Compares the worst case performance of the checkpointing protocol and our message logging protocol for different ratios of overhead (r) of the message logging protocol. The y-axis plots the total execution time normalized by $t_o(1 + \frac{d}{c})$. The x-axis plots the mean time between failures (m). The execution time for the message logging protocol with values of overhead 0%, 10%, 20%,50%,100% is shown.

Figure 6.3 is used to illustrate with an example exactly how the overhead of our protocol and the mean time between failure on a machine can affect the decision to choose between our fault tolerance protocol and a checkpointing

based protocol. Figure 6.3 plots the total execution time normalized by $t_0(1+\frac{d}{c})$ against the mean time between failure. It does so for an application that is assumed to checkpoint once every hour ($c = 1\ hour$) for both protocols and has 16 objects per processor in the case of our fault tolerance protocol ($v = 16$). Moreover, the constant overhead for restarting is assumed to be the same for the checkpoint and message logging protocols and is assumed to be 3 minutes ($k_c = k_{ml} = .05hours$). This is actually a conservative assumption since the checkpoint protocol has to restart all the processors and needs to fetch all their checkpoints, whereas the message logging protocol needs to get only one checkpoint. We plot the total execution time for the message logging protocol for multiple values of overhead(r), namely 0%, 10%, 20%,50%,100%.

The first thing to note in Figure 6.3 is that the time for the checkpoint increases sharply once the mean time between failures starts approaching the time between checkpoints (1 *hour*). This does not happen for the calculated time for the message logging protocols. The execution time remains reasonable even when failures are more frequent than checkpoints. Their execution time increases sharply only when the time between failure starts approaching their recovery time (close to 7 *minutes*).

Next, we see that if our message logging protocol does not impose any overhead (or a negligible one) on an application, then our message logging protocol performs better than a checkpoint based one even when failures are rare. This is borne out by the line for $r = 1.0$ which is always lower than that for t'_c. For higher overheads, our protocol continues to perform better than a checkpoint based protocol for certain ranges of fault frequency. Exactly at which frequency of faults, the checkpoint based method becomes better for an application depends on the overhead of our protocol for that application. The higher the overhead for an application, the lower the time between failures at which our

protocol starts out performing the checkpoint based one. This point is the one at which the line for t'_{ml} for a given overhead (value of r) crosses the one for t'_c. From this analysis, we see that our fault tolerance protocol will out perform a checkpoint based one in high fault frequency regimes. It can allow an application to make progress when it becomes impossible for a checkpoint based one. However, even if the fault frequency is low our fault tolerance protocol is better for applications on which it imposes lower overheads.

Chapter 7

Experimental Results

We evaluate the performance of the recovery protocol in both the basic and fast recovery modes. We also characterize the applications that are most suitable for our message logging protocol. We investigate the different performance penalties paid by various applications while using our protocol. Optimizations to reduce these overheads are also presented and evaluated.

We tested different parts of our protocol on 4 different machines:

- The **Architecture** cluster is a cluster of 16 dual Opteron (Processor 244) machines with 1 GB of memory, connected by Gigabit switched ethernet. We used gcc 4.0.1 and gfortran as the C++ and fortran compilers respectively.

- **Uranium** is a cluster of 20 dual processor 1 GHz Pentium III processors with 1.5 GB of memory and swap each. The nodes are connected with Gigabit switched ethernet and myrinet. We used gcc 3.3.5 as the C++ compiler.

- **Abe** is a cluster of 1200 nodes connected by Infiniband and Gigabit ethernet. Each node consists of 2 quad-core 64 bit 2.33 Ghz Intel Clovertown processors. We used the icc 10 suite of compilers for C and C++.

- **Tungsten** is a cluster of 1280 compute nodes connected by Gigabit ethernet and myrinet. Each node is a dual processor 3.2 Ghz 32-bit Intel Xeon processor.

7.1 Restart Performance

Virtual processors per processor	Basic Restart Time(s)	Fast Restart Time(s)
2	28.45	18.31
4	28.21	13.45
8	28.17	9.57
16	29.37	7.58

Table 7.1: Comparison of restart performances on 16 processors

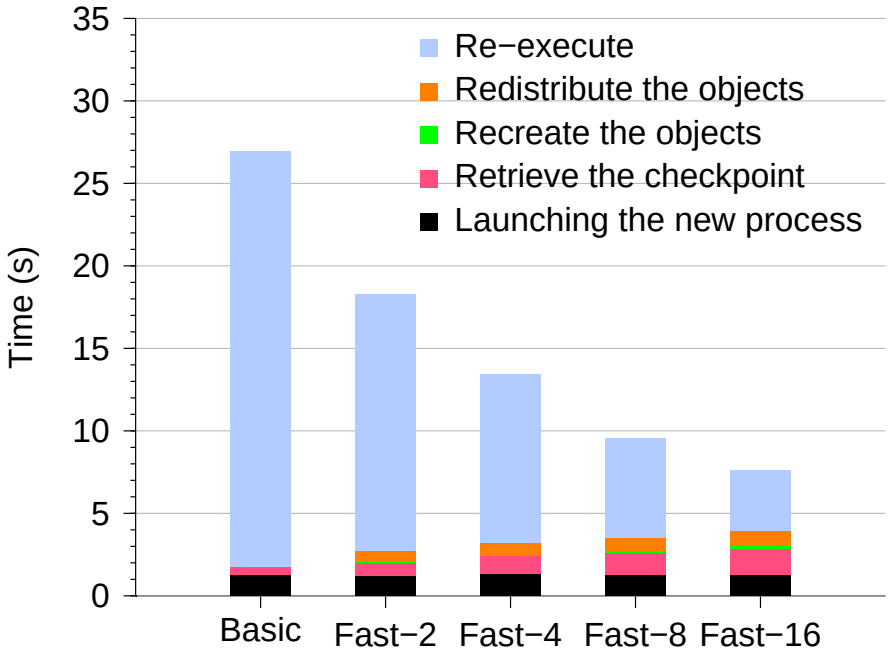


Figure 7.1: Different phases of the Basic and Fast restart protocols. The basic protocol was run with 1 virtual processor per processor. The fast restart protocol shows the times for 2,4,8 and 16 virtual processors per processor.

We use a 7-point stencil with 3D domain decomposition written in MPI to evaluate the performance of the restart protocols. In each iteration a Charm++ object gets data from its neighbors on all 6 sides and performs some computation. We ran the stencil code with two versions of AMPI, one with (*AMPI-FT*) and the other without (*AMPI*) the fault tolerance protocol. In the case of

AMPI-FT we checkpointed every 30 seconds. Our aim with this application is to evaluate the performance of the restart protocols. We simulate a fault on a processor by sending SIGKILL to a process running on it. After a processor crashes and restarts, the objects on the surviving processors wait for the objects on the restarted processor to catch up. The amount of time each object waits shows up as an increase in the run time of the ongoing iteration. We use the maximum increase in iteration runtime over all the surviving objects as a measure of the restart time for both the basic and fast restart protocols.

Table 7.1 shows the time taken for basic and fast restart for different numbers of virtual processors per processor. We ran the stencil code on 16 processors and checkpointed every 30 seconds. For each run a fault was triggered about 27 seconds after a checkpoint. Higher numbers of virtual processors per processor allowed the fast restart to distribute work among more processors and led to significantly shorter restart times. Even having just two objects per processor reduces the restart time significantly. Fast restart thus lets us restart much faster than the time between the crash and the previous checkpoint.

Recovery Phase	Basic	Fast-2	Fast-4	Fast-8	Fast-16
Launch New Process	1.29 s	1.24 s	1.34 s	1.27 s	1.28 s
Retrieve Checkpoint	0.44 s	0.76 s	1.05 s	1.31 s	1.55 s
Recreate Objects	0.05 s	0.09 s	0.12 s	0.17 s	0.21 s
Distribute Objects	NA	0.65 s	0.71 s	0.81 s	0.91 s
Re-execute	25.18 s	15.57 s	10.23 s	6.01 s	3.64 s

Table 7.2: The exact time spent in different phases of the restart protocol for Figure 7.1. The basic restart protocol was run with 1 virtual processor per processor. The fast restart protocol shows times for 2,4,8 and 16 virtual processors per processor.

Figure 7.1 compares the time spent in different phases of the basic and fast restart protocols. We measure the time taken to launch a new process, retrieve its checkpoint, recreate the objects and distribute them among other

processors. We subtract the total time taken during these four stages from the overall restart time to obtain the time taken to re-execute the work lost due to the crash. For the fast restart protocol, the re-execution time includes the time taken to receive the recovering objects on other processors. Table 7.2 shows the exact time spent in different parts of the restart protocols. The basic restart case was run with 16 objects on 16 processors and the fast restart protocol was run with numbers of objects per processor varying from 2 to 16.

The time to launch a new process is constant across the different runs. The overhead for retrieving the checkpoint increases with increasing number of objects because the checkpoint size grows with the number of objects. Some of the increase in checkpoint size is caused by data structures that are constant for every object such as the AMPI thread's stack and protocol data structures. Moreover, in a stencil application the amount of communication (in bytes not just number of messages) increases as we decompose the same domain into more pieces. Since the number of physical processors remains constant, this means that the sum of the size of the message logs of all objects on a processor increases with the number of objects. As message logs are a part of an object's state in the modified protocol, the checkpoint size increases with the number of objects on a processor. Therefore, the time to retrieve the checkpoint rises as the number of objects per processor increases. In fact, with 16 virtual processors per processor it becomes a significant part of the total restart time, taking about 20.4% of the total restart time.

The cost of recreating the objects is low and although the cost increases with the number of objects, it remains a small part of the overall restart time even when there are 16 virtual processors per processor. The overhead of redistributing the objects across different processors increases slowly with the number of objects per processor. Larger numbers of objects per processor means that ob-

jects are distributed among more processors and the fast restart protocol sends out more messages. When there are 16 virtual processors per processor, the cost to redistribute the objects forms about 12% of the total restart time.

However, the re-execution time decreases sharply with increasing number of objects per processor as the work of the restarted processor gets distributed among more processors. For the fast restart protocol with 16 virtual processors per processor the time to re-execute is a fraction of the basic restart protocol. In fact in this case, the overheads (first four rows in Table 7.2) associated with the fast restart protocol add up to more than the re-execution time itself. Thus the overall restart time starts to get limited by the increasing restart overheads. However, the decrease in re-execution is still far more than the rise in restart overheads due to higher numbers of objects. As a result, with larger numbers of objects per processor the fast restart protocol can recover much faster than the basic restart.

We also found that the forward path overhead (ie. overhead in absence of faults) for the stencil application was around 10% for the 16 processor run (a more detailed analysis of the forward path cost is presented in Section 7.2). Thus, our protocol provides the stencil application with fast recovery without imposing an unacceptably high performance cost.

We compared the recovery times of the basic and fast restart protocols for varying time durations between a crash and its previous checkpoint. We ran the 3D stencil MPI application with 512 virtual processors on 32 processors of the uranium cluster and triggered faults at varying times after a checkpoint. Figure 7.2 shows the recovery time while using the basic and fast restart protocols. As the time between the crash and the previous checkpoint increases, the amount of work that needs to be redone also increases. For the basic restart protocol, all the work is re-executed on the recovering processor. Therefore, the recovery

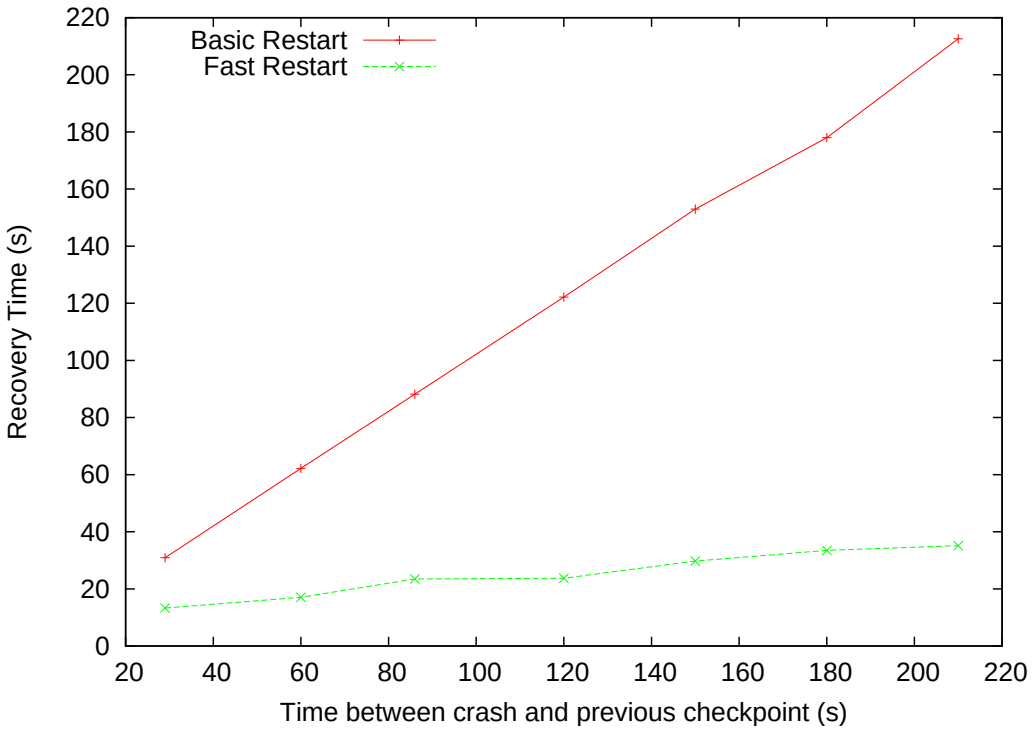


Figure 7.2: The Recovery time for the Basic and Fast restart protocols for different time durations between the crash and the previous checkpoint.

time is more or less equal to the time between the crash and the previous checkpoint. So, the line for the basic restart protocol has a slope more or less equal to 1.

The fast restart protocol speeds up recovery by distributing the work of the recovering processor among other processors. Its recovery time is a fraction of the time between the crash and its previous checkpoint. So even when there is more work to be done during recovery, its recovery time rises far less sharply than that of the basic restart protocol. Therefore, the line for the fast restart protocol is flatter in Figure 7.1. Moreover, as the time between a crash and its previous checkpoint increases, the recovery time for the fast restart protocol becomes a smaller fraction of that of the basic restart protocol. This happens because the overheads of the fast restart protocol (identified earlier in this sub-

section) form a smaller fraction of the total recovery time as the time duration between a crash and its previous checkpoint increases.

Thus, we see that the fast restart protocol speeds up recovery compared to the basic restart protocol in different situations. The recovery process of the fast restart protocol does have some extra overheads compared with the basic restart protocol. However, the advantage derived by parallelizing the re-execution of work during recovery far outweighs the extra costs of fast restart.

7.2 Application Studies

We want to characterize the applications that are most suitable to our message logging protocol. We want to evaluate the overhead of our protocol for different types of applications. We expect the increase in message latency to be the main source of overhead. We found that, on our Opteron cluster, short message latency for AMPI is 45 μs and that for AMPI-FT is 125 μs. This is exactly what we expect with the short message round trip (twice message latency for AMPI), needed to log the meta-data of a message, accounting for this difference. We use the NAS parallel benchmarks to categorize the types of applications that would suffer large or small performance penalties in the face of this increased message latency. We run NPB3.1 with versions of AMPI with and without the fault tolerance protocol. We show data for the class B of four representative benchmarks : CG, MG, SP and LU. For a particular number of physical processors, we run each benchmark with varying numbers of virtual processors and report the best performance for AMPI. In the case of AMPI-FT we report the performance for different numbers of virtual processors on a certain number of physical processors. As we are trying to measure the overhead of the message logging protocol, we do not take any checkpoints during the execution of the

benchmarks.

Figure 7.3 shows the performance of the MG benchmark on a varying number of physical processors. Since performance is measured in terms of Mflops, a higher bar represents better performance. For each physical processor we show the performance of AMPI-FT with 1,2,4 and 8 virtual processors per processor. For a small number of processors the performance penalty of our protocol is low when using 1 virtual processor per processor. On a small number of processors, the application is mostly computation dominated and the increase in communication time due to the increased message latency does not affect overall performance that much.

However, on higher numbers of processors the increased message latency starts to make its effects felt. AMPI-FT with just 1 VP per processor suffers a severe performance penalty. Figure 7.3 show that this penalty can be mitigated by adding more virtual processors per processor. The performance of MG on AMPI-FT improves sharply with higher degrees of virtualization. This happens because having more virtual processors per processor enables adaptive overlap of communication and computation. While one virtual processor is waiting for a message, another virtual processor on the same processor can continue with its execution. This allows MG to effectively hide the increased latency due to the message logging protocol. The number of virtual processors needed per processor to achieve the best performance varies between 4 and 8 for MG class B on this range of processors. This represents a compromise point between the divergent performance impacts of virtualization. Too few virtual processors means that there is not enough adaptive overlap of communication and computation. On the other hand, having too many virtual processors increases the number and volume of communication for a particular application. The best performance occurs where there is a good degree of adaptive overlap

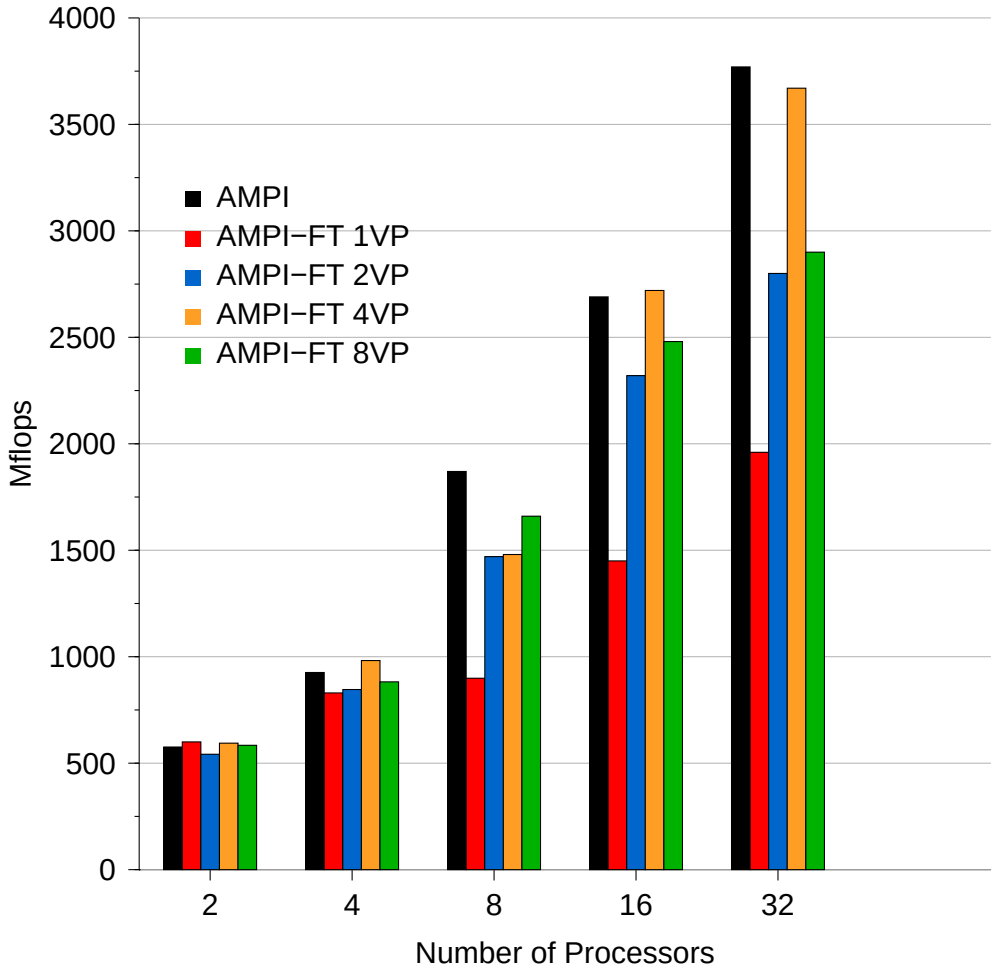


Figure 7.3: Performance of the MG class B benchmark. The AMPI FT values are shown for different numbers of vp per processor.

but the benefits of adaptive overlap have not been wiped out by the increased communication costs.

Figure 7.4 shows the performance of the SP benchmark on AMPI and AMPI-FT. The SP benchmark is designed so that it can only run on a square number of virtual processors. This meant that for a certain number of physical processors, there were only a few possible values for total number of virtual processors that were a square and yielded an integral number of virtual processors per processor. So, we simply ran the SP benchmark with 1 virtual processor per

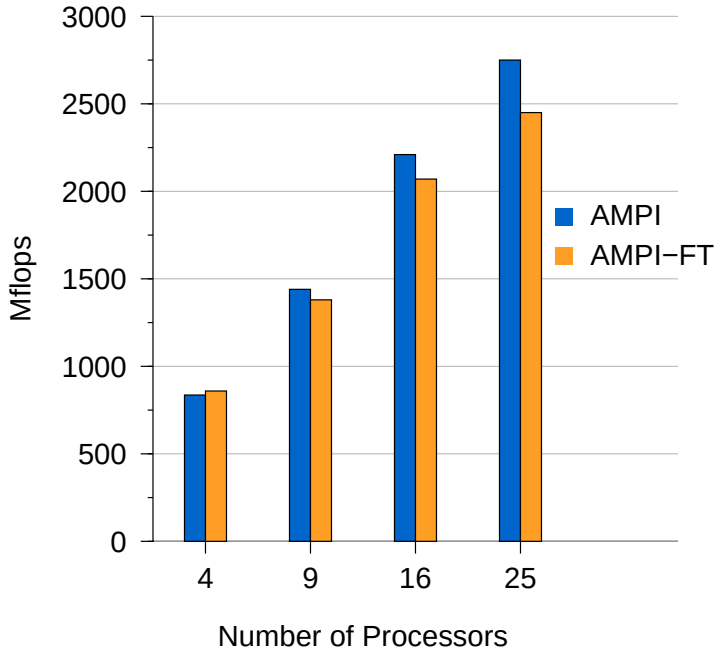


Figure 7.4: Performance of the SP class B benchmark.

processor. In this case, we found that AMPI-FT had a low performance penalty even without the benefit of virtualization.

The performance of the CG benchmark is shown in Figure 7.5. Increasing the number of virtual processors per processor, improves the performance of CG even on small numbers of processors. CG, as we shall see a little later, is more communication intensive than MG. Therefore, even on small numbers of processors, adaptive overlap of computation and communication helps performance. The performance penalty of CG is low until 16 processors. On 32 processors, having 2 virtual processors per processor improves performance over the 1 virtual processor per processor case only a little. With higher number of virtual processors, the performance in fact deteriorates. This happens because on 32 processors the amount of work per processor is low for CG and there is not sufficient computation to overlap with the increased communication la-

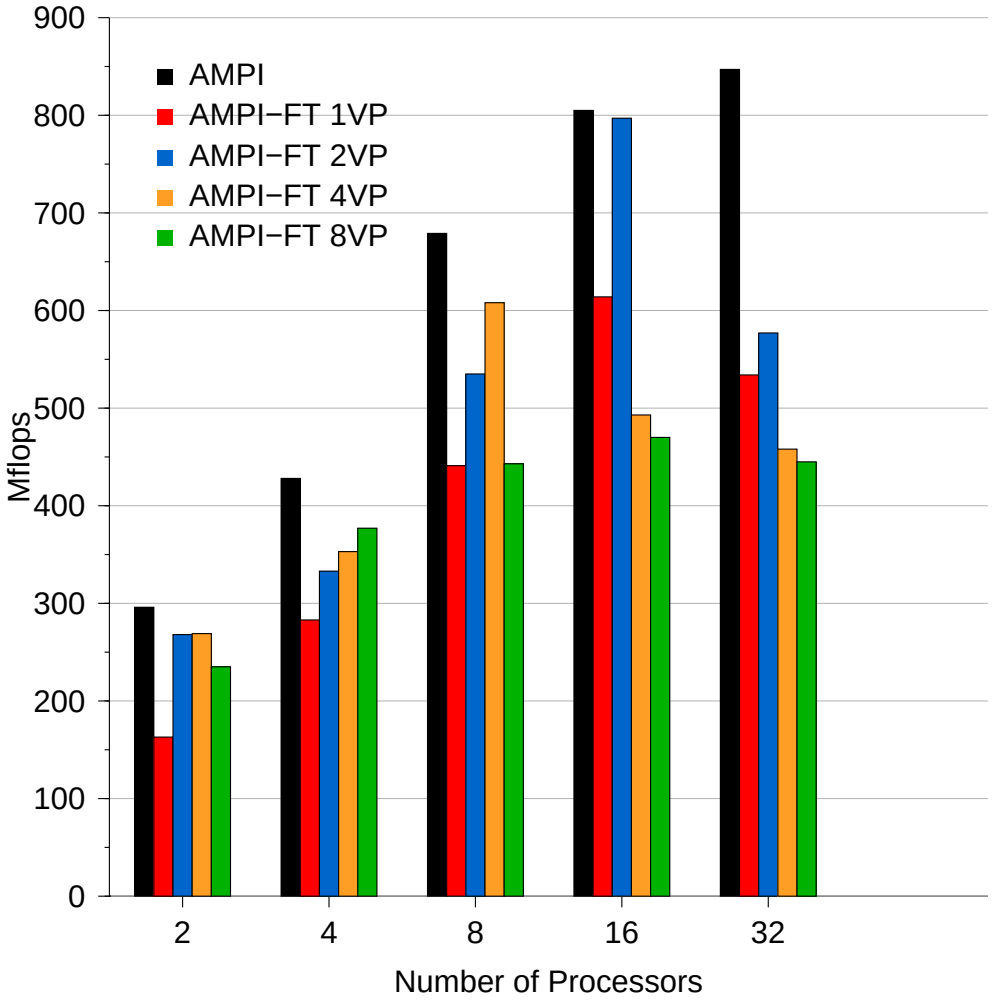


Figure 7.5: Performance of the CG class B benchmark. The AMPI FT values are shown for different numbers of vp per processor.

tency. Moreover, increasing virtualization adds communication load of its own. As a result, CG's performance on AMPI-FT deteriorates with higher numbers of processors. One thing to note about CG is that even on AMPI, without the fault tolerance protocol, the performance of CG does not scale well beyond 16 processors.

Figure 7.6 shows that the LU benchmark performance. The performance penalty is low for small numbers of processors with virtualization moderating

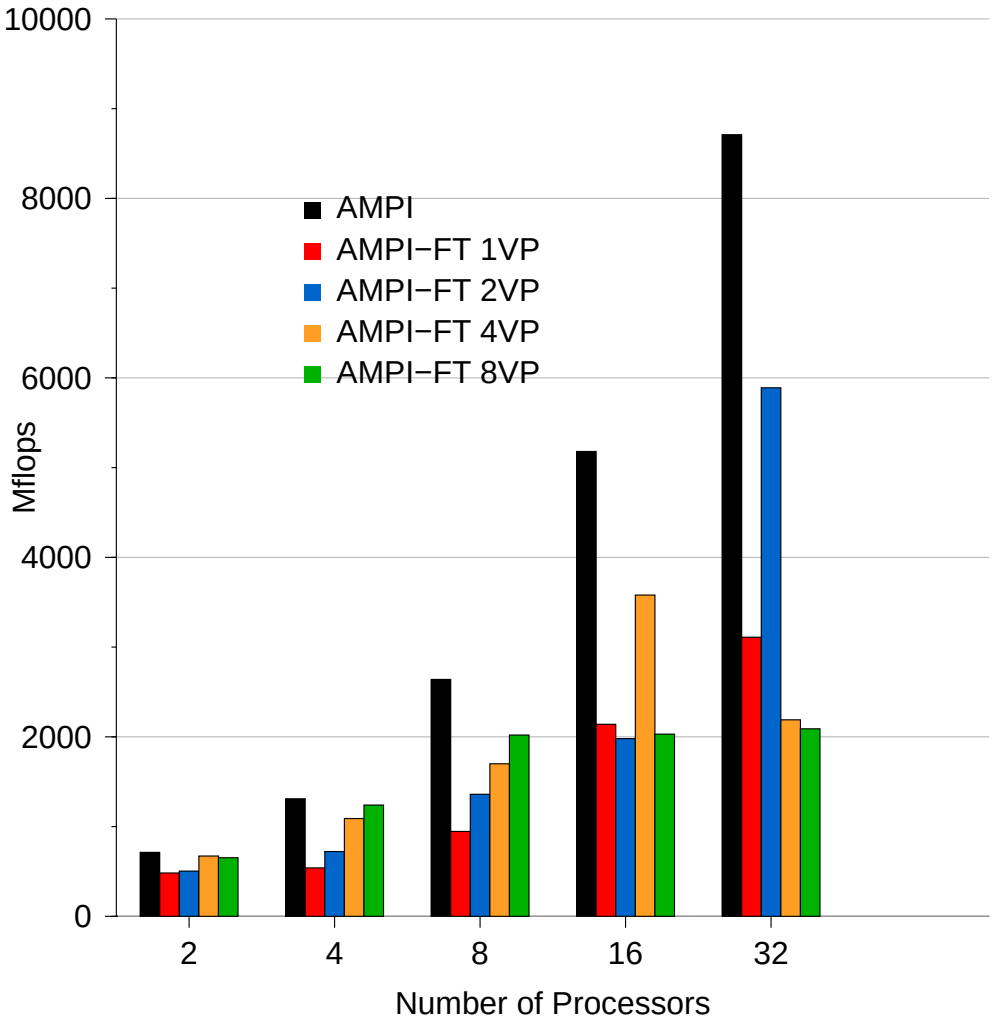


Figure 7.6: Performance of the LU class B benchmark. The AMPI FT values are shown for different numbers of vp per processor.

the overhead of the increases message latency. However, LU pays a significant performance penalty while using AMPI-FT on larger numbers of processors. Having more virtual processors per processor, decreases the penalty significantly even on large numbers of processors. However, even after the improvement the penalty for LU is high (33% on 32 processors).

Figures 7.3 and 7.4 show that the performance penalty is low for the MG and SP benchmarks respectively. The performance penalty for CG in Figure

7.5 is moderate, whereas that for LU in Figure 7.6 is significant. MG's good performance is expected since it sends the least number of messages[25, 52]. For the same class of problem and number of processors, MG sends about a quarter the number of messages as any of the other benchmarks. Similarly LU's bad performance is expected as LU sends almost 5 times as many messages as any of the other 4 benchmarks. The situation gets a bit confused when we consider CG and SP's performance. We know that for the same class and number of processors SP actually sends a larger number of messages than CG [25]. The different performance penalties imposed by AMPI-FT on each benchmark can be explained if we consider the number of instructions executed per message sent by each benchmark. We find that the MG and SP benchmarks execute about the same number of instructions per message sent [20]. Both the benchmarks execute about a couple of million instructions per message send on 16 or more processors. On the other hand LU and CG execute a few (less than 6) hundreds of thousands of instructions per message sent. This means that the increase in message latency forms a smaller fraction of the computation time per message for MG and SP than for LU and CG. So the overall performance penalty is lower for MG and SP. SP has a higher performance penalty compared to MG since SP sends more and larger messages than MG [20].

In Figures 7.7 and 7.8 we look at the CPU overheads associated with different parts of the message logging protocol for the MG and LU benchmarks respectively.. Both benchmarks were run on 32 processors, MG with 128 virtual processors and LU with 64. These were the best configuration for each benchmark on 32 processors. The time spent by the CPU in different phases of the protocol is expressed as a percentage of the runtime while using AMPI. In addition to the protocol components, the time spent in computation as well as the time that processors were idle are also shown. The percentages for the

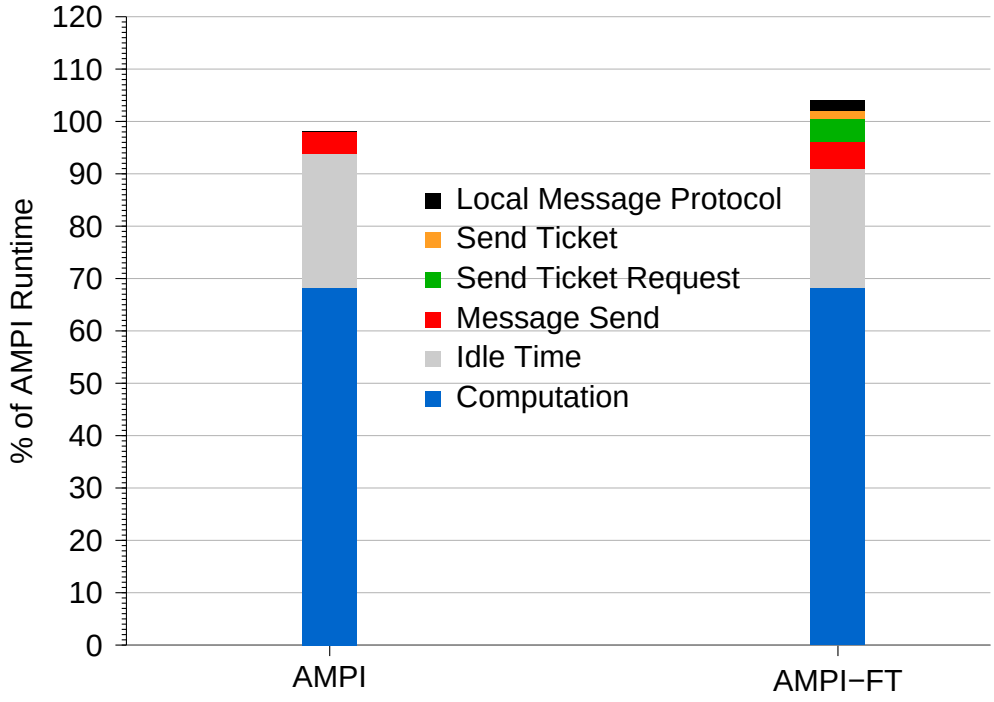


Figure 7.7: Break up of the execution time in the case of AMPI and AMPI-FT relative to the total AMPI runtime for MG on 32 processors. We use the number of virtual processors that was the best for both runs. The number of virtual processors per processor for AMPI was 1, whereas it was 4 for AMPI-FT.

different components are averaged over all the processors.

Figure 7.7 shows that for MG the computation time, as expected, remains unchanged while using our message logging protocol. The idle time actually shows a decrease while using the message logging protocol. This happens because some of the computation related to the protocol gets overlapped with the time spent waiting for communication. So, a processor which would have been idle in AMPI utilizes a part of that time in AMPI-FT. The message send time is marginally higher in the case of AMPI-FT since the protocol requires some book keeping when a message is being sent as a result of receiving a ticket. Sending ticket requests takes up about the same time (4% of AMPI runtime) as sending messages in the case of MG. Sending ticket requests and the local mes-

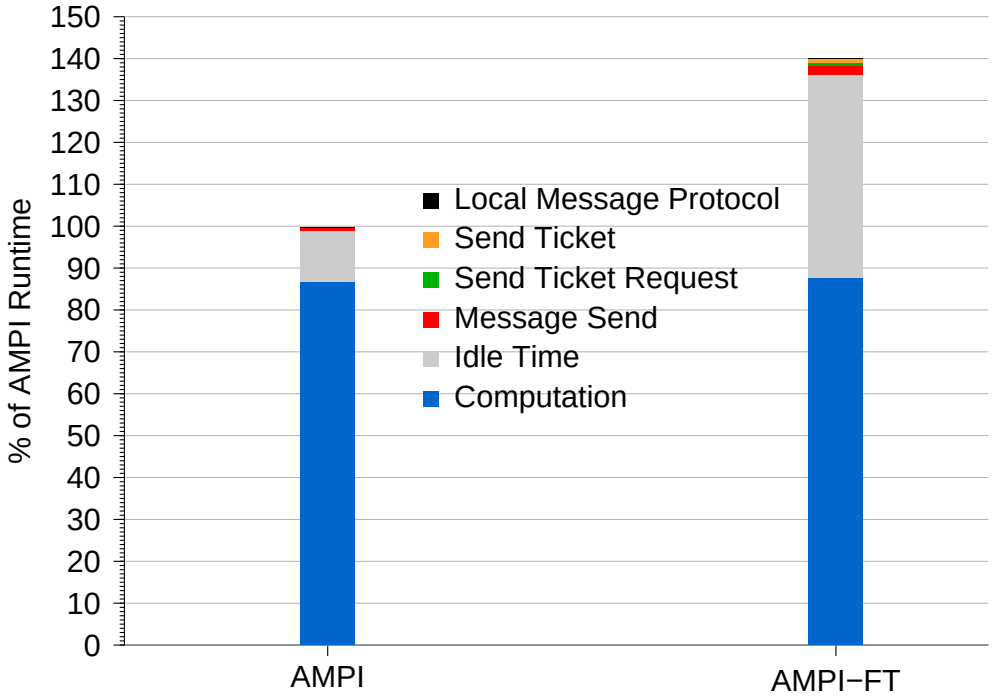


Figure 7.8: Break up of the execution time in the case of AMPI and AMPI-FT relative to the total AMPI runtime for LU on 32 processors.We use the number of virtual processors that was the best for both runs. The number of virtual processors per processor for AMPI was 1, whereas it was 2 for AMPI-FT.

saging protocols consume only a small amount of CPU time. These small CPU overheads of the different parts of the message logging protocol are the primary source of overall increased execution time in the case of MG. The MG benchmark can hide the increase in message latency by overlapping the computation of one virtual processor with the communication of another. However, the increased CPU usage due to the message logging protocol can not be overlapped completely with the idle time and ends up increasing the overall execution time.

The LU benchmark in Figure 7.8 presents a situation very different from that of the MG benchmark. Although, the computation time remains the same for AMPI and AMPI-FT, the idle time sees a substantial increase when the fault tolerance protocol is being used. The increased message latency due to message

logging means that objects have to wait longer for messages. LU manages to overlap some amount of computation with this waiting time and thus partially mitigate the penalty of increased latency. This is borne out by the fact that in Figure 7.6 the performance of LU with AMPI-FT on 32 processors is much better when there are two virtual processors per physical processor rather than one. However, LU has such a fine granularity with low instructions per message that there is just not enough work to overlap with the increased message latency. Attempts to increase the amount of possible overlap between communication and computation by increasing the number of virtual processors also fail beyond a certain point because dividing the work into more pieces just increases the number of messages and reduces the granularity further. The cost of these overheads outweigh any benefits of overlapping communication and computation. Thus, in the case of the fine grained LU benchmark the performance penalty of the fault tolerance protocol is mostly caused by the increase in idle time due to higher message latency.

7.3 Protocol Overhead for Different Application Granularity

We found in Section 7.2 that the performance penalty imposed by the message logging protocol on an application depended on the number of instructions executed per message. We use a synthetic benchmark to take a closer look at the relationship between performance and the number of instructions executed per message.

The synthetic benchmark is a very simple iterative MPI program. The MPI processors are logically arranged in a ring. In every iteration, each MPI process sends a short message each to its neighbors on the left and right. Each

MPI process also receives a message from each of its neighbors on the left and right. After that, every MPI process performs some calculations for a specified amount of time. The amount of time spent in the calculation in each iteration is a measure of the *granularity* of the application. Although, this measurement of granularity is not the same as the number of instructions, the two are closely related, particularly since the computation loop repeatedly performs the same calculations.

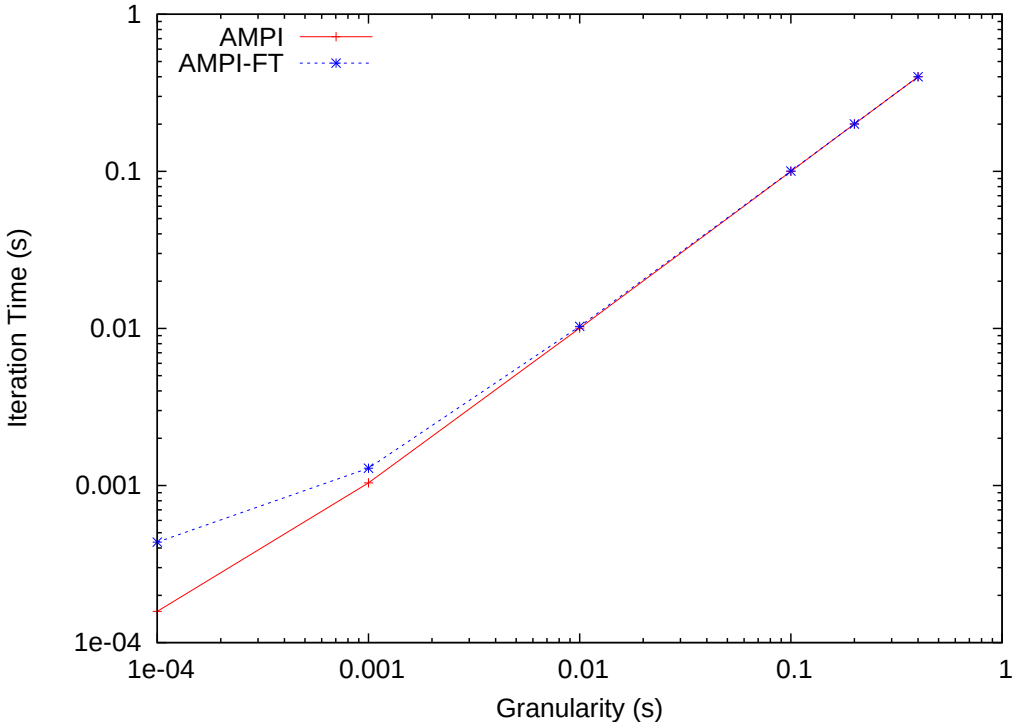


Figure 7.9: Iteration time against granularity for AMPI and AMPI-FT with 8 virtual processors on 8 physical processors

We tested the synthetic benchmark on the Tungsten cluster using the myrinet interconnect. In the first experiment, we evaluated the synthetic benchmark with 8 virtual processors on 8 physical processors for both AMPI and AMPI-FT. We varied the granularity from 100 μs to 400 ms. Figure 7.9 shows the iteration time for different values of granularity for both AMPI and AMPI-FT.

We see that the performance penalty is insignificant when the granularity is more than equal to 10 ms. For the lower values of granularity, the increased message latency FT and the CPU overhead of the protocol impose a high performance penalty. Since there is one virtual processor per physical processor in this experiment there is no adaptive overlap between communication and computation to hide the increased message latency.

Next, we tested our hypothesis that increasing the number of virtual processors per processor helps us reduce the performance penalty of our protocol. We ran the synthetic benchmark with 32 virtual processors on 8 physical processors. We break up the same amount of work into more pieces. So, each virtual processor now does a quarter of the work done in the previous experiment. The amount of work per physical processor remains the same. Figure 7.10 shows the iteration time for different granularities when there are four virtual processors per processor. The performance penalty for high values of granularity is negligible as in the previous experiment. More importantly, the performance penalty for the low granularity cases is lower when there are more virtual processors per processor. The 1 ms granularity case sees the performance penalty decrease by 40% when we have 4 virtual processors per processor instead of 1. Increasing the number of virtual processors improves the performance for the 100 μs case as well. However, the improvement is small and even with 4 virtual processors per processor the iteration time for AMPI-FT is nearly 2.5 times that of AMPI. It seems to suggest that for very low granularities, increasing the number of virtual processors introduces an additional source of overhead that nearly cancels out the benefit of adaptive overlap of computation and communication. This additional source of overhead is associated with the message logging protocol since the performance of AMPI actually improves when the number of virtual processors is increased (158 μs with 1 vp and 142 μs with 4 vps).

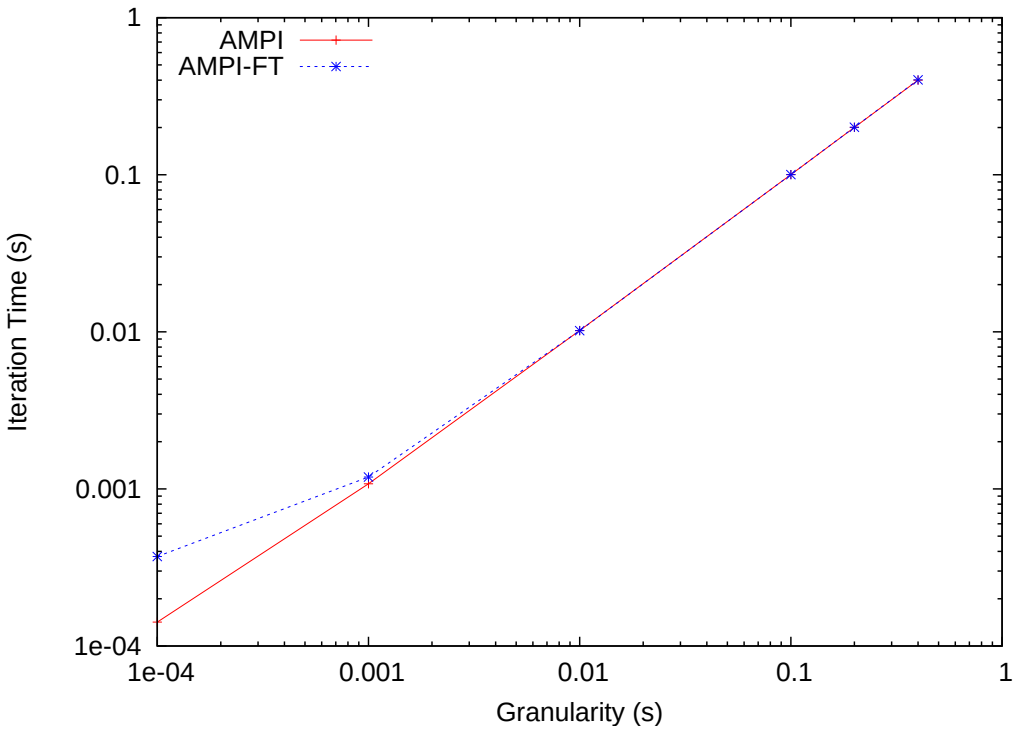


Figure 7.10: Iteration time against granularity for AMPI and AMPI-FT with 32 virtual processors on 8 physical processors

One major difference between AMPI and AMPI-FT is that messages between virtual processors on the same physical processor result in messages to an external processor in the case of AMPI-FT but not in the case of AMPI. In AMPI-FT, the meta-data for a message between virtual processors on the same processor needs to be saved on the buddy processor before the message can be processed. This difference can lead to a much higher number of messages being sent on the network for AMPI-FT than AMPI. We measured the number of messages being sent on the network in every iteration of the synthetic benchmark for AMPI and AMPI-FT. The measurements were performed with 32 virtual processors on 8 processors in both cases. In the case of AMPI, each physical processor sent 2 messages on the network per iteration. For AMPI-FT, that number went up to 18 messages per iteration. Of the 18 messages sent per

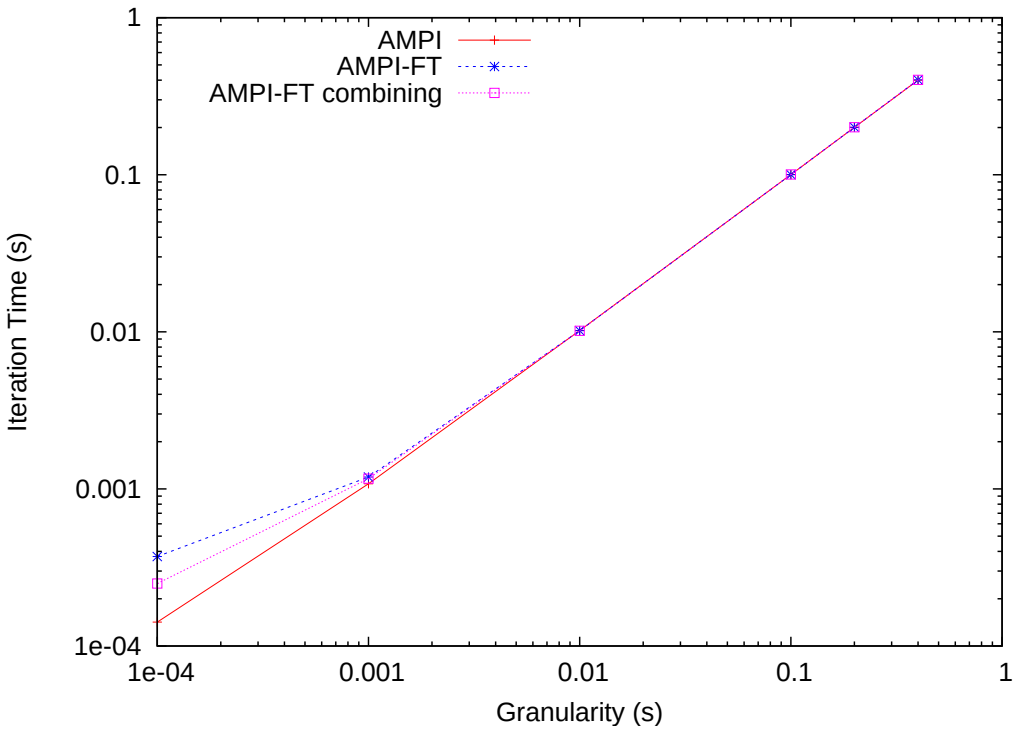


Figure 7.11: Iteration time against granularity for AMPI, AMPI-FT and AMPI-FT-combining with 32 virtual processors on 8 physical processors

iteration, 6 were meta-data being sent to the buddy to be saved and another 6 were acknowledgements of the receipt of these TNs. This meant that in the 100 μs granularity case, each processor was trying to send 180000 messages per second for AMPI-FT instead of 20000 messages per second for AMPI. Moreover, 120000 of these messages per second were exchanged between a processor and its buddy. We suspect that this deluge of messages stressed the network and its performance degraded sharply.

We ran a pingpong program to test myrinet's performance at high message rates. We measured the round trip time between two processors at different message rates. We found that the round trip time for a message increased sharply when more than 60000 short messages were exchanged per second between two processors. So, we concluded that the very large number of messages being

exchanged by a processor and its buddy in the 100 μs case degrades AMPI-FTs performance.

This leads us to try and reduce the number of messages being exchanged by a processor and its buddy. We merged multiple meta-data being sent to a buddy into one message. We also merged multiple acknowledgements being sent by the buddy into one message. The number of meta-data being merged into one message is configurable by the user. The user can also choose a timeout period such that when the timeout expires, the processor stops waiting for more meta-data and sends to the buddy the meta-data it has accumulated till then. This does not affect the ability of the protocol to handle faults since the protocol does not depend on how meta-data are delivered to the buddy or how long it takes them to reach the buddy. As long as the buddy gets the meta-data, stores it and sends an acknowledgment, the protocol's ability to tolerate faults is not affected.

Figure 7.11 shows the performance of AMPI-FT when message combining is used. AMPI-FT with combining shows a significant performance improvement over AMPI-FT for the 100 μs case. When there are 4 virtual processors per processor, the performance penalty for AMPI-FT with combining is 53 % lower than that for AMPI-FT. Message combining also decreases the performance penalty for all the other values of granularity as well.

7.4 Optimizations and their Effects on Application Performance

The previous section showed that the performance of fine grained programs with our message logging protocol can be improved by temporarily buffering message meta-data being sent to a buddy and combining the buffered meta-data

into one message. We also used only one message to send the acknowledgements for all the meta-data that arrived in one message. Moreover, when the buddy relationship is symmetric, meta-data can be piggybacked on returning acknowledgements and vice-versa.

7.4.1 Fine Grained Application

We now evaluate the effectiveness of this idea when dealing with a real fine grained application. We run the very small *BUTANE* molecular system on leanMD to evaluate the idea of combining multiple meta-data messages required for the local protocol into one message. For this evaluation, we run the protocol in the single simultaneous failure mode to clearly isolate the effect of the local and remote modes of the protocol. There are two configurable parameters in our scheme to buffer meta-data being sent to a buddy:

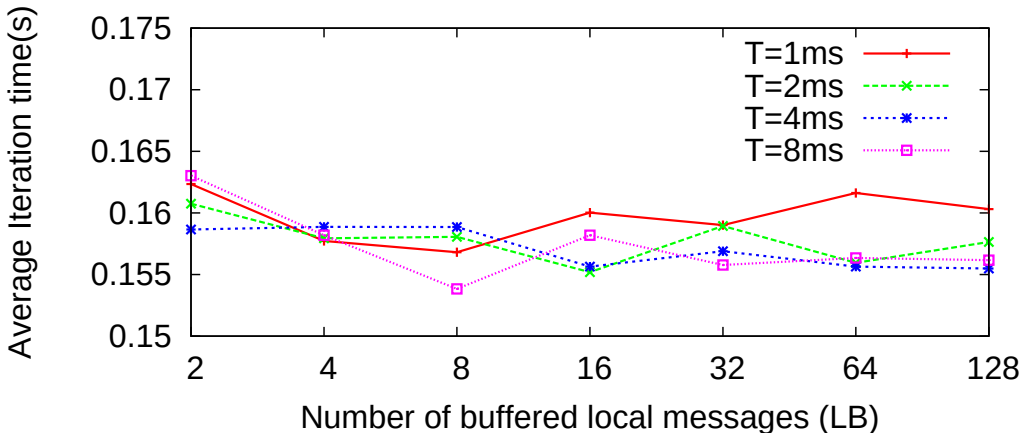
- The maximum number of local mode messages whose meta-data are combined into one message to the buddy processor. We refer to this as the number of buffered local messages (LB).
- The time-out duration after which buffered meta-data are sent to the buddy even if the number of buffered messages is less than the allowed maximum. We simply call this the time-out (T)

Buffering meta-data for local mode messages has multiple effects, some beneficial and some harmful. The number of messages on the network can be reduced by buffering meta-data. This improves network performance and reduces message latency. This in turn can reduce the amount of idle time during which processors wait for messages. Moreover, buffering multiple meta-data into one message reduces the per meta-data CPU cost on both the sending and receiving processors. Combining multiple meta-data and acknowledgment

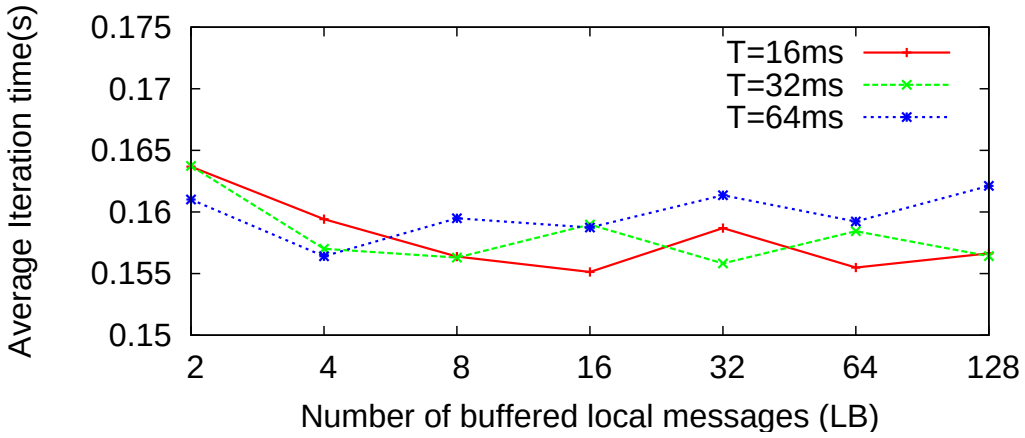
messages helps amortize the fixed portion of the CPU overhead of sending a message. Thus buffering multiple local message meta-data leads to a reduction in the CPU overhead of the local mode of the protocol.

On the other hand, buffering meta-data for local messages delays the processing of those messages, since a local message can not be processed until the buddy processor acknowledges that it has saved the meta-data for that message. This delayed processing can potentially boost the idle time by further increasing message latency. This is particularly true when we use long time-out durations with large buffer sizes. We end up waiting for long periods of time just waiting for the buffer to fill up with meta-data being sent to the buddy. This hurts performance and can cancel out potential benefits of combining meta-data messages for the local mode of the protocol.

We try to evaluate what parameters for LB and T are most suitable for leanMD simulating the the fine grained BUTANE molecular system on 16 processors of Abe. On 16 processors, each processor sends about 6600 messages per second when leanMD simulates BUTANE without using the fault tolerance protocol. We use only 16 processors for this example since BUTANE is too fine grained a problem to scale beyond that. We show the results on a bigger molecular system on a larger number of processors later. The average iteration time for BUTANE without fault tolerance is .055 s on 16 processors. The average iteration time with fault tolerance but no buffering on 16 processors is .164 s. So, the performance penalty of the fault tolerance protocol without any buffering is extremely high (about 200%). It increases execution time to 3 times, close to the worst case scenario discussed in Section 5.2. We now evaluate the efficacy of buffering meta-data for the local mode of the message protocol. Figure 7.12 shows the average iteration time for different values of LB and T.



(a) Iteration time for time-outs equal to $1ms$,$2ms$,$4ms$,$8ms$



(b) Iteration time for time-outs equal to $16ms$,$32ms$,$64ms$

Figure 7.12: The average iteration time for the BUTANE molecular system in leanMD on 16 processors. We vary the number of message meta-data buffered as well as the time-out for the buffer.

Figure 7.12(a) shows the performance for lower values of time out durations. We are limited to using a time-out duration of $1ms$ as the smallest time-out by the resolution of the timer in the CHARM++ run-time system. We find in Figure 7.12(a) that T=1 ms does not improve performance much for any value of LB. A time-out of 1 ms is not sufficiently long enough to buffer a large number of messages. The cases for $T = 2$ ms and $T = 4$ ms show better performance. However, the best iteration time is obtained when $T = 8$ ms and $LB = 8$. At this point, the gap between the benefits of buffering meta-

data and their drawbacks is the largest. Higher values of LB do not improve performance since they force the time-out to occur thereby wasting time and increasing idle time. Figure 7.12(b) shows the performance for higher values of time out durations. They show some performance improvement for low sizes of local buffer. However, with large sizes of buffer their performance deteriorates as time-outs become frequent leading to an increase in idle time.

Buffer size	Time out Duration (ms)	Total Local mode time(ms)	Protocol time / local message (μs)	Total Idle Time(s)
Unbuffered	NA	36.330	28.84	3.041
8	8	11.607	9.21	2.914
16	2	16.857	13.38	2.936
128	64	5.637	4.48	3.176

Table 7.3: Compares the total time and per message time spent in the local mode of the message logging protocol for different local message buffer sizes and time out durations. These times pertain to a 30 iteration run. The idle time for the different values are also shown.

Table 7.3 can be used to understand how buffering meta-data for local messages affects performance of leanMD simulating BUTANE on 16 processors. We look at the total time spent in the local mode of the protocol, the per local message time spent in the local mode of the protocol as well as the total idle time for a run lasting 30 iterations. We instrumented our message logging protocol code to obtain this detailed breakup. We used only 30 iterations to limit the amount of performance data generated while still getting a representative picture of the application's performance. We looked at the breakup for 4 runs: unbuffered, the best with $LB = 8$ $T = 8$ ms, another well performing one $LB = 16$ $T = 2$ ms and the worst of the buffered runs $LB = 128$ $T = 64$ ms. All the buffered runs spend, in the local mode of the protocol, only a fraction of the time spent by the unbuffered run. The per local message time is also much lower for these runs. The run with the lowest values is the worst performing one with the largest

buffer and longest time. This apparent anomaly can be explained by looking at the total idle time for each run. The 4^{th} entry, ie the worst buffered run, has a significantly higher idle time than the other buffered runs. This increase in idle time for the worst buffered run is much higher than its savings in the local mode of the protocol. This is in line with what we discussed earlier in this section. The best times are obtained by runs which reduced not only the local time but also the idle time. These runs managed to do so by distributing the fixed cost of sending a message among multiple meta-data and acknowledgments without wasting too much time waiting for new local message meta-data.

Although, buffering meta-data of local messages improves performance, the improvement is marginal. It decreases average iteration time from .164 s to .0.154 s, compared to a time of .055 s without fault tolerance. Therefore, we need to look into where the extra time is being spent. Figure 7.13 shows the amount of time spent in different phases of the protocol for different values of LB and T. We use the same runs and in the same sequence as Table 7.3.

The most obvious thing about Figure 7.13 is that the time spent in the local mode of the message protocol is a very small portion of the total time spent in the message protocol. The time spent in the local mode of the protocol does indeed decrease for the buffered cases. However, any improvements are dwarfed by the time spent in the remote mode of the protocol. This explains why even the best runs with local meta-data buffering show only a slight improvement in performance.

It is imperative to reduce the time consumed by the remote mode of the protocol if we want to improve the overall performance of the application with the fault tolerance protocol. The remote mode of the protocol sends short messages to request tickets and send tickets back to the requesters. The BUTANE benchmark on 16 processors has every processor sending about 4100 ticket requests

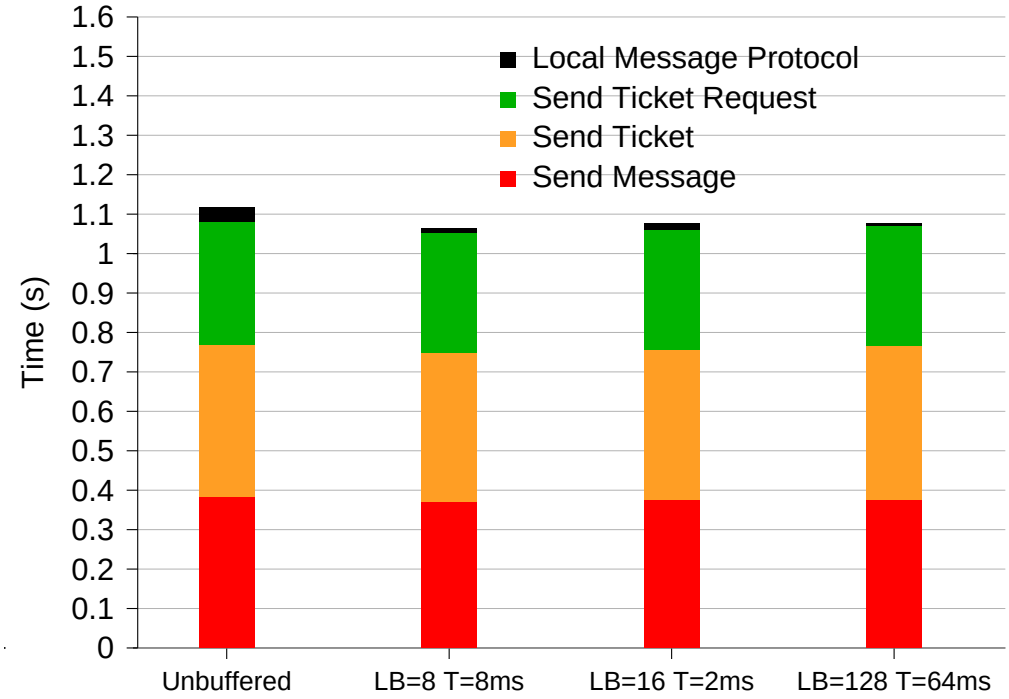
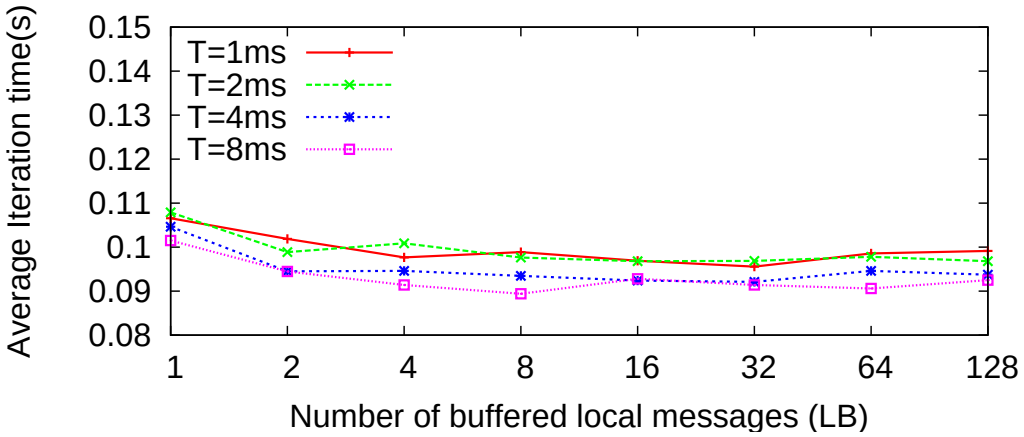


Figure 7.13: The total time spent in different parts of the message logging protocol for different values of local buffer size and time out duration. *LB* refers to buffer size for local messages, *T* refers to the time out duration.

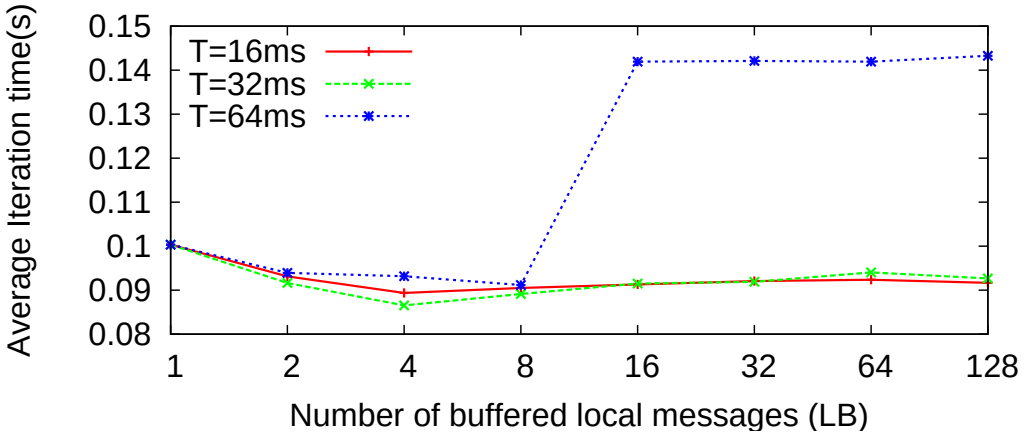
per second. on average. Each processor needs to reply to a similar number of ticket requests and send tickets in reply. Thus, BUTANE on leanMD creates a flood of small messages in the remote mode of the protocol as well. We extend the same idea of combining multiple small messages going to the same processor into one message to reduce the overhead of the remote mode of the protocol. A major difference between message combining for the local and remote modes is that in the case of remote mode a processor sends small protocol messages to multiple processors and not just the single buddy processor.

We buffer the protocol messages for the remote mode being sent to different processors. Messages to each processor are buffered separately. Moreover, these buffers are created for a processor only when there are protocol messages being actually sent to that processor. The maximum number of protocol messages

being combined into one message is configurable and the parameter is called the remote buffer size (RB). We re-use the time-out duration parameter from the local meta-data buffering scheme. There is a separate timer for every destination processor. When a timer for a particular destination processor runs out, all the buffered protocol messages destined for that processor are sent. Buffered messages for other processors are not affected .



(a) Iteration time for time-outs equal to $1ms$,$2ms$,$4ms$,$8ms$



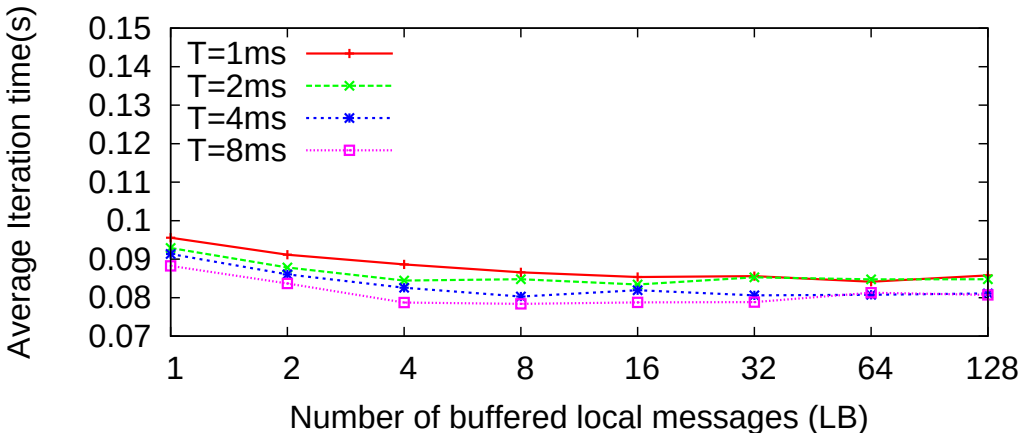
(b) Iteration time for time-outs equal to $16ms$,$32ms$,$64ms$

Figure 7.14: The average iteration time for the BUTANE molecular system on 16 processors with at most 4 remote protocol messages being combined into one protocol message. We vary the number of local messages being buffered as well as the time-out.

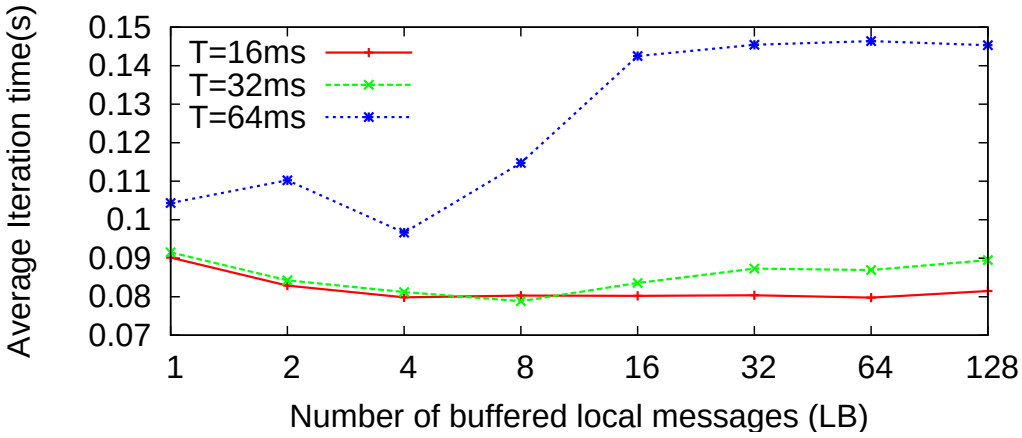
We evaluate the performance benefit of buffering remote mode protocol messages. Figure 7.14 shows the performance of leanMD simulating the BUTANE

molecular system on 16 processors when $RB = 4$. With at most 4 protocol messages between two processors being combined into one message, we vary the number of local meta-data being combined into one message as well as the time-out duration. We see that buffering remote protocol messages provides a substantial improvement in performance. It reduces the average iteration time from .154 s with local meta-data buffering to .0.0894 s with $RB = 4$ $LB = 4$ and $T = 32$ ms. This sharp decrease in average iteration time holds for most values of LB and T tested in Figure 7.14. If we look at the case where $LB = 1$, ie no local meta-data are being buffered, we still see a substantial performance improvement. For high values of T we get average iteration times slightly less than .1 s. This confirms that the performance improvement in Figure 7.14 is mostly due to remote mode protocol message buffering. Moreover, it should be noted that, without local buffering, increasing the value of T reduces the average iteration time till $T = 32$ ms. This means that for the BUTANE system, the improvement in performance obtained by reducing the remote mode protocol time through higher time-out outweighs the increase in idle time.

Varying the LB and T produces improvements of the same order as in Figure 7.12. Moreover, for most values of T, average iteration time decreases with increasing LB, till the range of 4 to 16 before starting to increase slowly for larger values of LB. The performance of the $T = 64$ case deteriorates sharply when the number of buffered local message is increased to 16 and beyond. We investigate this in greater detail later. However, we can safely guess that the very high time-out duration and high local buffer size combine to increase the idle time as processors wait for meta-data messages to be acknowledged. For lower values of time-out, the cost of waiting for more buffered messages is low. When the buffer size is lower, the processor does not have to wait as long for the buffer of local message meta-data to fill up.



(a) Iteration time for time-outs equal to $1ms$,$2ms$,$4ms$,$8ms$



(b) Iteration time for time-outs equal to $16ms$,$32ms$,$64ms$

Figure 7.15: The average iteration time for the BUTANE molecular system on 16 processors with at most 8 remote protocol messages being combined into one protocol message. We vary the number of local messages being buffered as well as the time-out.

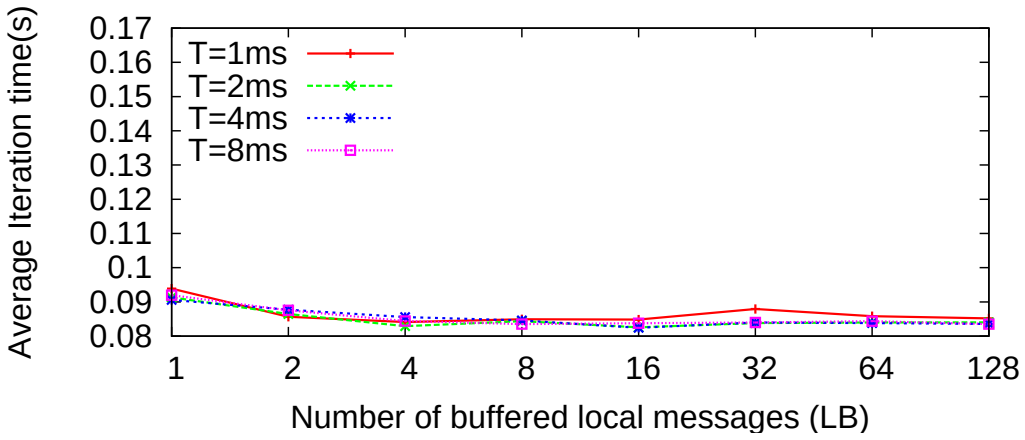
Figure 7.15 shows the performance of leanMD simulating BUTANE when $RB = 8$. We found the greatest performance improvement with remote protocol message buffering in this case. The average iteration time is lowest, 0.0784 s, when $LB = 8$ and $T = 8$ ms. The performance curve is similar for all values of T less than 16 ms. In all these cases, even without any local message buffering ($LB = 1$) there is a significant improvement in performance over the case without any remote protocol message buffering. The performance improves as LB is increased till a value between 4 and 16. In this range, more buffering reduces

total time spent in the remote mode protocol without increasing the idle time too much. After that, the time wasted by waiting for more and more messages increases the idle time and wipes out the benefits of buffering.

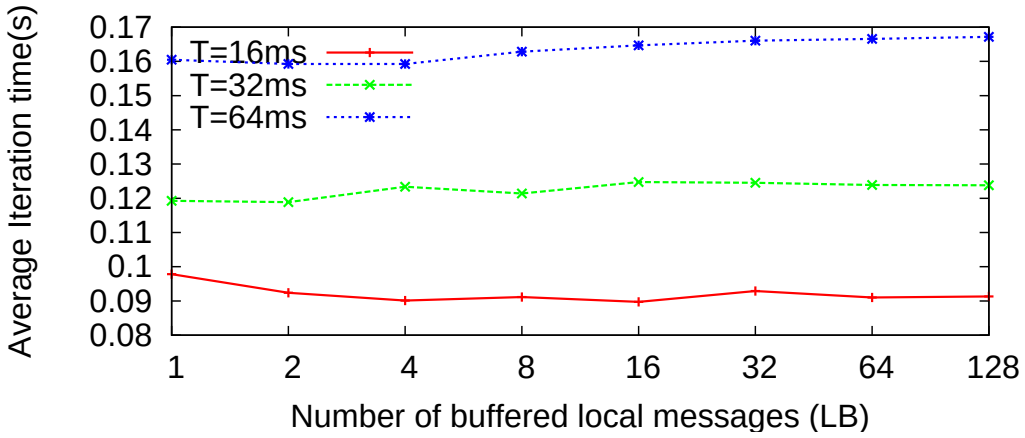
It is interesting to note that the difference in average iteration time between different values of T for the same value of LB is more or less constant across all values of LB. It is particularly true for T between 1 ms and 16 ms. This difference can be attributed to the difference in performance improvement achieved by buffering remote protocol messages for varying time-out durations. Increasing T increases the efficacy of the remote buffering until $T = 8$ ms. After that the increase in idle time caused by longer waits starts to cancel out benefits of buffering. The $T = 64$ ms case is a stark example, where a lot of time is wasted while waiting for time-outs so that remote protocol messages and local meta-data can be sent.

Figure 7.16 shows the performance achieved when $RB = 16$. For, small values of T it shows performance similar but a little worse than the $RB = 8$ case. Its performance for these moderate values of T is better than the $RB = 4$ case. It shows a pattern similar to the previous cases, with increasing LB improving performance initially before it starts to negatively affect performance. However, the situation is very different for higher values of T. Higher values of T mean that when there are not enough messages to fill up the buffer, we end up waiting for longer before sending those messages. Since with $RB = 16$, we wait for more messages than the $RB = 4$ or $RB = 8$ cases, we also end up waiting longer. For low values of T the benefit of combining more messages is close to the possible cost of increased idle time. However, when T is high the increase in idle time is far higher than any reduction in protocol time.

Figure 7.17 helps us evaluate the effectiveness of buffering just the protocol messages for the remote mode. Figure 7.17 plots the average iteration time



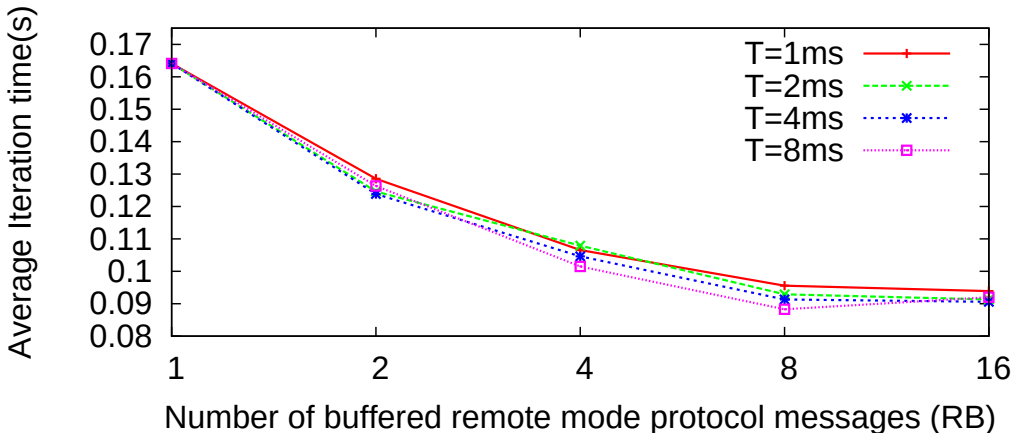
(a) Iteration time for time-outs equal to $1ms$,$2ms$,$4ms$,$8ms$



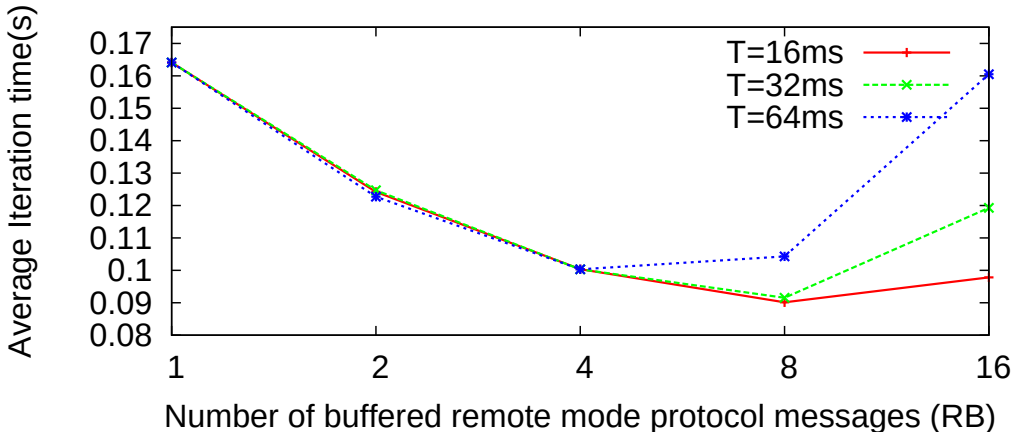
(b) Iteration time for time-outs equal to $16ms$,$32ms$,$64ms$

Figure 7.16: The average iteration time for the BUTANE molecular system on 16 processors with at most 16 remote protocol messages being combined into one protocol message. We vary the number of local messages being buffered as well as the time-out.

against RB for different values of time-out (T). Buffering of meta-data of local mode messages is turned off at this point. We see that buffering remote protocol messages improves performance significantly for all time-out durations. For short time-out durations, performance keeps improving till $RB = 8$ and then starts to level off. The reduced idle time and CPU protocol overhead produced by buffering overshadows any increased latency to reduce the average iteration time. In fact the best time with local buffering turned off, 0.088 s, is obtained when $RB = 8$ and $T = 8$ ms. Increasing the buffer size further does not



(a) Iteration time for time-outs equal to $1ms$,$2ms$,$4ms$,$8ms$



(b) Iteration time for time-outs equal to $16ms$,$32ms$,$64ms$

Figure 7.17: The average iteration time for the BUTANE molecular system on 16 processors with varying values of RB and time-out(T). There is no local buffering ($LB = 1$).

help since there are not enough remote protocol messages and the time-out gets triggered repeatedly. As a result, the number of remote protocol messages combined into one network message does not increase. So, no further benefit accrues from increasing the buffer size. In fact for large time-out durations, the increased latency due to messages being buffered for longer starts to overpower the advantages of buffering. For $T = 64$ ms, the average iteration time starts rising beyond $RB = 4$. Thus, we see that buffering protocol messages for the remote mode just by itself helps performance greatly, particularly for certain

ranges of RB and T.

Figure 7.18 looks at the amount of time spent in different parts of the message logging protocol as well as the idle time for various values of RB, LB and T. The times shown are for 30 iterations of BUTANE on 16 processors. We show data for 5 cases: the unbuffered case, the best performing one with $RB = 8$ $LB = 8$ $T = 8$ ms, another well performing one with $RB = 8$ $LB = 4$ $T = 8ms$, a worse performing one with $RB = 4$ $LB = 16$ $T = 64ms$ and one of the worst with $RB = 16$ $LB = 128$ $T = 64ms$.

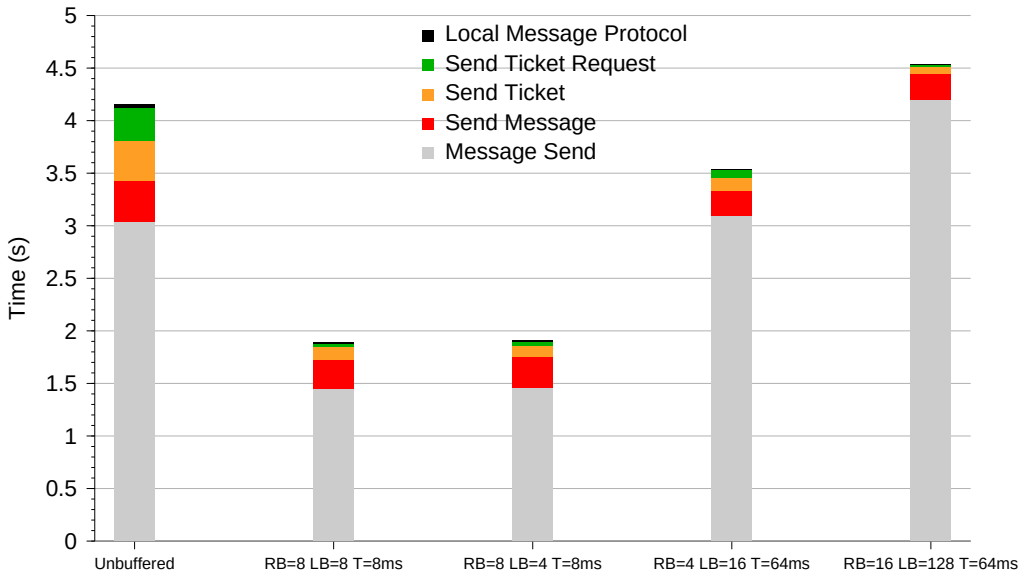


Figure 7.18: The total idle time as well as the total time spent in different parts of the message logging protocol for different values of remote buffer size, local buffer size and time out duration.

All the runs with buffering show a substantial decrease in the time spent in portions of the message logging protocol when compared to the unbuffered run. The decrease in the time taken for sending ticket requests to remote objects and the local mode of the protocol are particularly sharp (almost a factor of 10). The time taken to send tickets decreases less sharply but still by a factor of about 3. This is so because the CPU time needed to send a short message is a bigger

fraction of sending a ticket request than of sending a ticket. Generating a ticket consumes more CPU-time than generating a SN. So, amortizing the cost of a short message send by combining multiple protocol messages into one message is even more effective when sending a ticket request than while sending a ticket. The cost of sending messages decreases too but only by about 25%. We do not combine the actual data messages being sent by the user code since that would lead to an unfair comparison with the non fault tolerant version. Still there is an improvement because now multiple messages are sent out as the result of processing a single message containing multiple tickets from objects on a particular processor. This divides the fixed cost of processing a message among multiple messages sends. This is responsible for the observed improvement. These improvements are seen for all the buffered runs, even those with high average iteration times.

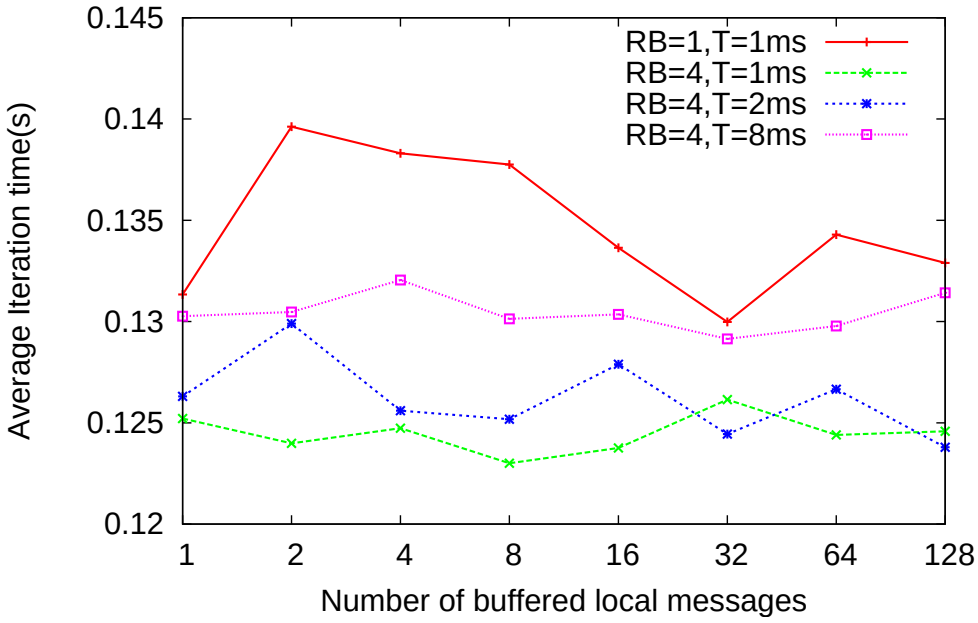
The biggest difference between the best performing runs and the bad ones is the idle time. For the 2^{nd} and 3^{rd} runs in Figure 7.18, the idle time is less than half that of the unbuffered run. Buffering remote protocol messages and local meta-data not only reduces the protocol overhead but also decreases the time wasted waiting for messages. This shows up in the halved idle time. However, for the last two runs, their high time-out durations mean that if there are not sufficient messages to fill up a particular buffer, those messages get delayed substantially before they are sent. This means that any object waiting to process those messages is also held up. This increased latency ends up keeping processors idle for longer durations. As a result, the idle time increases wiping out most of the benefits accrued by reducing the CPU time consumed by different parts of the protocol. The increase in idle time is sharp enough for the last run that it actually takes longer to run than the unbuffered run.

We used buffering for both remote protocol messages and local meta-data to improve performance of leanMD simulating BUTANE on 16 processors. The best performance of .0784 s was obtained with $RB = 8$,$LB = 8$ and $T = 8$ ms. This is a marked improvement over the unbuffered time of .164 s. Although the improved performance still represents a 42% overhead over the average iteration time of .055 s, obtained when not using the fault tolerance protocol, it must be remembered that the BUTANE molecular system represents a worst case scenario for our message logging protocol.

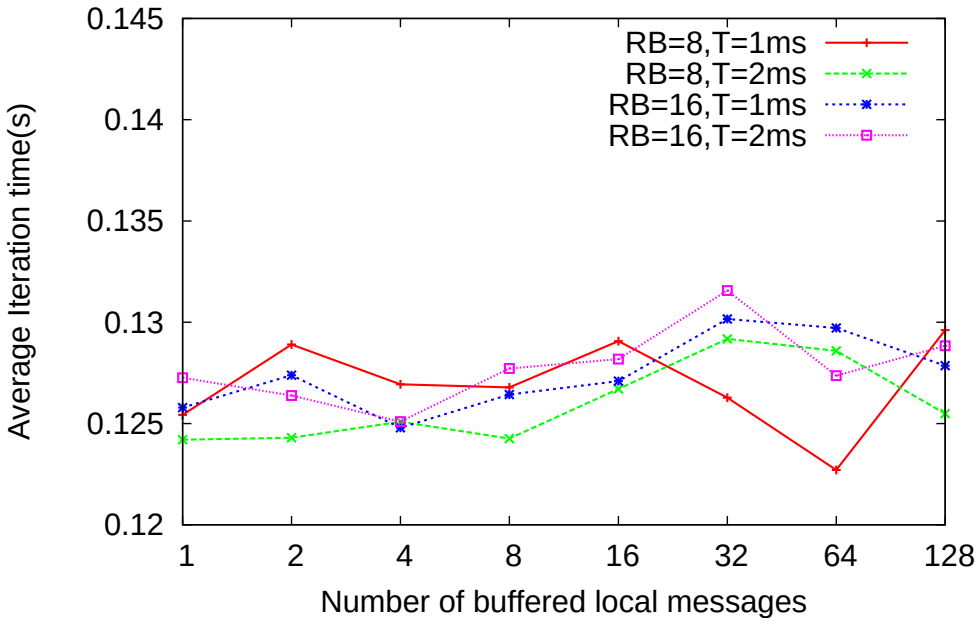
7.4.2 Coarse Grained Application

We next evaluated the performance penalty of our fault tolerance protocol on a more coarse grained application running on a much larger number of processors. We used leanMD to simulate a different molecular system called *HCA_GRP_SHAKE* . This molecular system is considerably more coarse grained than BUTANE. Therefore we were able to run it on 256 processors on Abe. Each processor sends only about 700 messages per second when leanMD simulates HCA_GRP_SHAKE on 256 processors. leanMD takes on average .112 s to perform one iteration of its simulation of HCA_GRP_SHAKE on 256 processors without the fault tolerance protocol.

When run with the fault tolerance protocol without any buffering, the average iteration time increases to .131 s. This represents a performance penalty of only about 17%. Thus the performance penalty for HCA_GRP_SHAKE without any optimizations is much lower than the close to 200% penalty faced by BUTANE without optimizations. So, as discussed in Section 7.2, applications with coarser granularities suffer far lower penalties. However, as shown in Figure 7.19, buffering remote protocol messages and local message meta-data sent to buddies can still improve performance.



(a)



(b)

Figure 7.19: The average iteration time for the HCA_GRP_SHAKE molecular system on 256 processors. We vary the number of local messages being buffered, the number of remote messages being buffered as well as the time-out.

Figure 7.19 shows the average iteration time for different values of LB, for certain values of RB and T. It plots the performance for the subset of collected

data point that showed the best performance. The point at which $LB = 1$ on the line for $RB = 1$ $T = 1$ ms represents the unbuffered case. When $LB = 1$, we find that increasing RB helps improve performance a lot. We get the best performance without local buffering when $RB = 8$ and $T = 2$ ms. The average iteration time at $LB = 1$ $RB = 8$ $T = 2$ ms is .124 s, which represents a 10% penalty over a run without fault tolerance. Thus, remote buffering just by itself helps reduce the performance penalty significantly.

Increasing LB improves performance a bit for some values of RB and T, particularly for $RB = 4$ $T = 1$ ms, $RB = 4$ $T = 2$ ms cases. leanMD simulating the HCA_GRP_SHAKE molecular system running on 256 processors has fewer local messages. Therefore buffering local messages does not yield as much benefit as for the BUTANE example. Still, it helps reduce the performance penalty in some cases. Overall, the best performance is observed when $LB = 64$ $RB = 8$ $T = 1$ ms, with an average iteration time of 0.122 s. This represents a 9% performance penalty for simulating the HCA_GRP_SHAKE molecular system on 256 processors. We think that this is an acceptable level of performance penalty for our fault tolerance protocol.

Figure 7.20 shows the total idle time and the total time spent in different parts of the protocol when leanMD simulated the HCA_GRP_SHAKE molecular system for 10 timesteps on 256 processors. We show data for 5 cases: 1) no remote protocol message or local meta-data buffering 2) the best performing example with local buffering but no remote buffering $LB = 32$ $RB = 1$ $T = 1ms$ 3) the best performing example with remote buffering but no local buffering $LB = 1$ $RB = 8$ $T = 2ms$ 4) the overall best performing example with $LB = 64$ $RB = 8$ $T = 1ms$ 5) the worst data-set that we measured $LB = 128$ $RB = 64$ $T = 2ms$.

When we compare the second bar in Figure 7.20 with the first one, we find

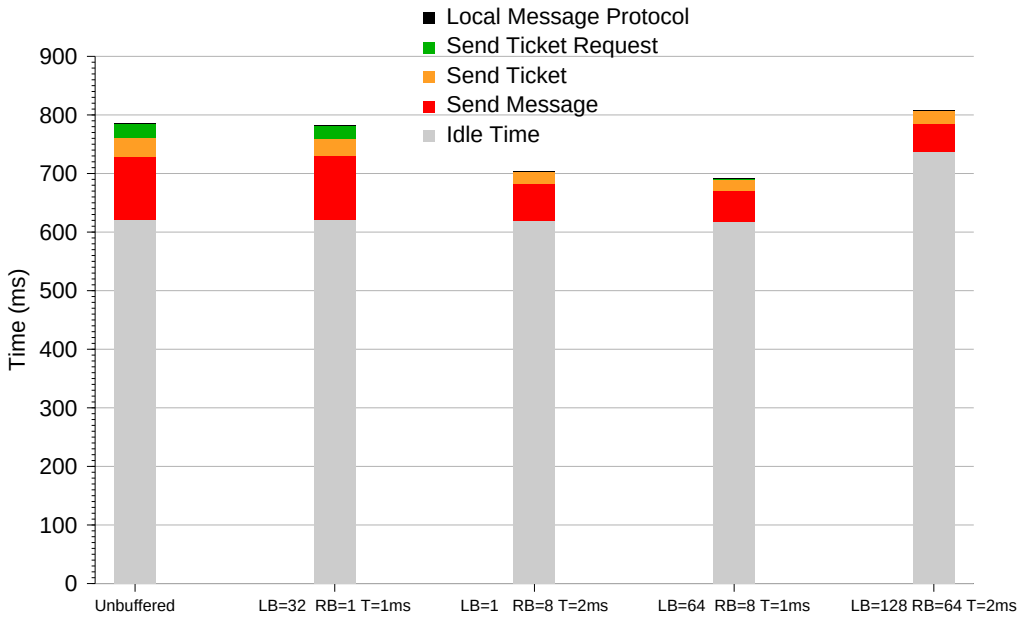


Figure 7.20: The total idle time as well as the time spent in different parts of the message logging protocol for different values of LB, RB and T during 10 timesteps of HCA_GRP_SHAKE

that for the HCA_GRP_SHAKE example local buffering makes a very small impact. This is in line with our observations in Figure 7.19. Although the time spent in the local mode of the protocol decreases, it is so small to begin with that the difference in the overall execution time is very small. Local buffering does not change the idle time much. The contrasting effects of increased idle time due to higher latency and lower idle time due to lower CPU protocol overhead cancel out.

Comparing the third bar in Figure 7.20 with the first one shows that buffering remote protocol messages reduces the time spent in the remote mode of the protocol significantly. The CPU overhead of sending and receiving messages on the network is amortized over multiple protocol messages by combining protocol messages into one message on the network. Particularly, the total time to send ticket requests goes down sharply. The time to send tickets and messages

also decreases. The total idle time decreases marginally due to the reduced CPU protocol overhead. However, most of the reduction in execution time is due to the reduced remote protocol overhead. The fourth bar, which represents the run with the overall best performance shows a performance similar to but slightly better than the third bar.

The last bar represents the worst performing run among the large set of runs we made. It also shows a reduction in protocol time for the remote mode. However, this benefit is completely wiped out by the increase in idle time. The high value of RB which leads to frequent time-outs increases the message latency. This causes the observed increase in idle time.

Thus we see that for a high granularity application with few local messages, buffering protocol messages in the remote mode of the protocol improves performance. This improvement in performance is caused by a decrease in CPU protocol overhead and not the decrease in idle time as in the case of a fine granularity application. There are multiple possible reasons for this. Firstly, as seen in Section 7.2 the idle time of a coarse grained application is relatively unaffected by our message logging protocol. As long as there are enough objects per processor, adaptive overlap of computation and communication helps hide the increased latency due to the message logging protocol. In a coarse grained application, unlike a fine grained application, there is enough computation to hide almost the entire increase in latency. Therefore, the scope for reducing idle time by buffering protocol messages is small. Secondly, in a coarse grained application the network is not as stressed by extra protocol messages as in a fine grained application. Therefore combining multiple protocol messages into a single message on the network does not provide as big a boost to network performance by relieving congestion. So in a coarse grained application combining messages does not reduce idle time as much. However, buffering does

help reduce the protocol overheads even for coarse grained applications and thus improve the overall execution time.

Therefore, we found that buffering protocol messages is an effective optimization technique to improve the performance of our message logging protocol. It improves performance for both fine and coarse grained applications. However, the improvement is more dramatic for fine grained application since there is more scope for improvement in such applications. These performance optimizations help reduce the cost of our fault tolerance protocol without affecting its ability to tolerate faults.

Chapter 8

Load Balancing With Message Logging

This chapter first shows why we need to be able to perform load balancing to take full advantage of the fast restart protocol. The CHARM++ run time system already has elaborate support for load balancing. In later sections of this chapter we shall see how the current load balancing system works and the challenges in combining it with a message logging based fault tolerance protocol. We shall then describe our solutions to those challenges.

8.1 Need for Load Balancing Along With Fast Restart

We shall now see how our fast restart protocol might lead to a load imbalance after the recovery of a crashed processor is complete. Let us consider as an example, the stencil application used in Chapter 7.1. We look at the case when there are 64 objects running on 16 processors. The amount of work on each object is about the same. The application is tightly coupled such that if an object stops making progress, then other objects stop within a short time as well.

The application is initially load balanced, that is there are 4 objects on each processor. When a processor, say P, crashes, the 4 objects on P are recreated from their previous checkpoint. The fast restart protocol keeps one object on

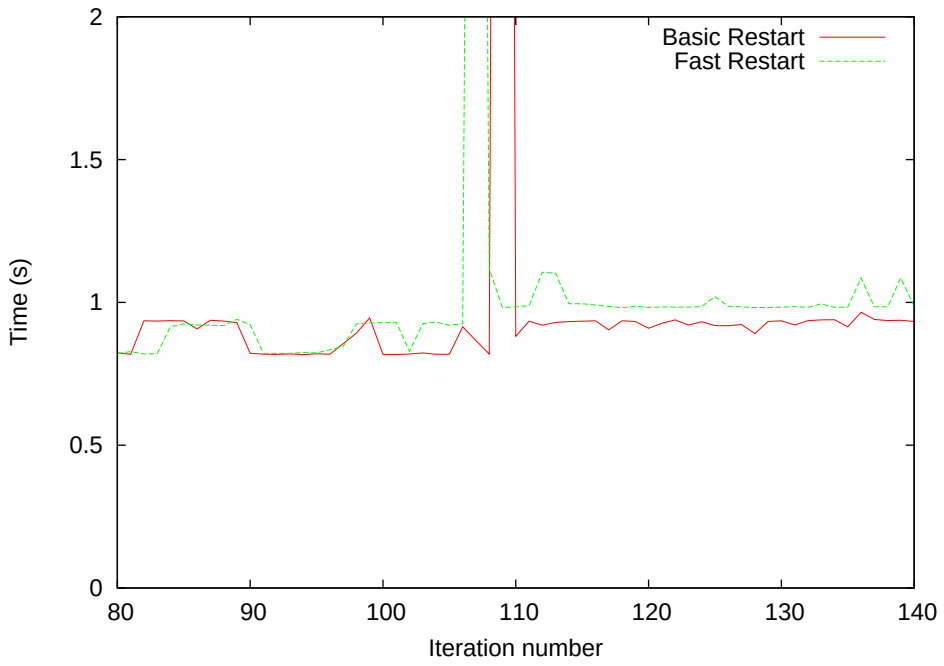


Figure 8.1: Iteration times for the 7-point stencil before and after a restart for both the basic and fast restart protocols

P and sends one object each to three other processors (let Q be one of these). The 4 objects originally on Q are waiting for the recreated objects to catch up with them. Until the recreated objects catch up with the rest of the application, the recreated object on Q has the entire processor to itself. However, once the recreated objects catch up and the application starts making progress, a new problem shows up. Now there are some processors with 5 objects, some with 4 and one (processor P) with just one object. This load imbalance increases the computation time for each iteration. Figure 8.1 shows the iteration time for the 7 point stencil program used in Section 7.1 before and after a restart for both the basic and fast restart protocols. The spike around iteration 100 denotes a processor crashing and recovering. It can be seen that in the case of the basic restart protocol, iteration times before and after the crash are more or less the same. However, for the fast restart protocol the iteration time after the crash

and recovery is distinctly higher than the iteration time before the crash.

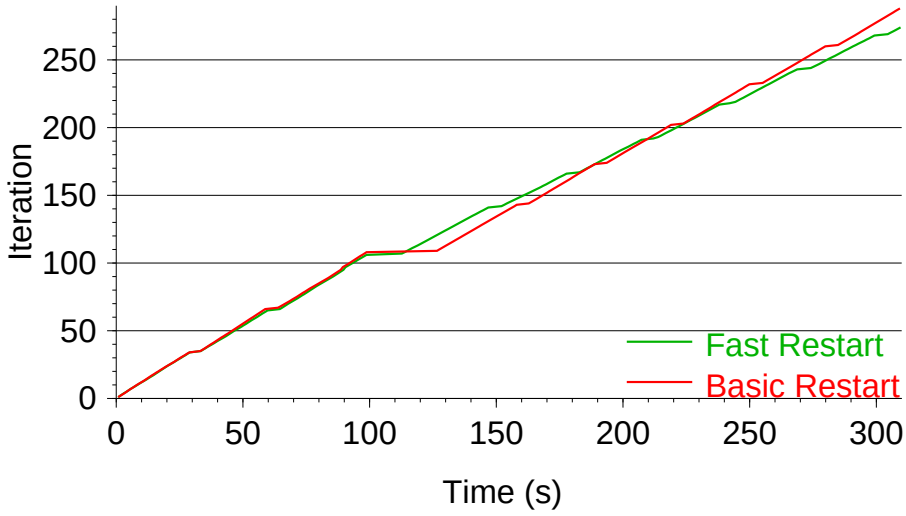


Figure 8.2: Progress of the 7 point 3D stencil application with both the basic and fast restart protocols. It shows that though the fast restart protocol shows faster recovery, it makes slower progress after the recovery.

Although fast restart improves recovery performance, it has a deleterious effect on the performance of an application after the recovery is complete. Figure 8.2 shows the effect of the performance penalty imposed by the fast restart protocol. For the same run as above, Figure 8.2 plots the iteration number against the cumulative time taken by the application to get to the end of that iteration. So, this graph shows the progress an application makes, with a steeper slope representing faster progress. We can see that until about the 100^{th} iteration both the fast and basic restart algorithms progress at the same rate. There is a crash around the 100^{th} iteration. At that time, the application stops computing further iterations and is busy recovering from the fault. This shows up as a flat horizontal section in the progress curve. This flat section is much longer for the basic protocol than the fast one. So, immediately after the recovery phase the fast restart progress curve is above the basic restart one. This shows that the

application started making progress sooner while using the fast recovery protocol. However, the basic restart progress line has a steeper slope, essentially the same slope as before the crash, and around the 220 iteration mark crosses the fast restart line. Thus the load imbalance caused by the fast restart protocol migrating objects during recovery can destroy the benefits of swift recovery from a crash.

8.2 Existing Load Balancing

The CHARM++ load balancing framework is a very flexible framework that is used to implement a wide variety of measurement based dynamic load balancing strategies [9, 17, 53, 36]. The framework measures the computation and communication load of the different CHARM++ objects and then utilizes this data to redistribute objects among processors to obtain a better load balance. The different strategies can broadly be divided into two categories: centralized and distributed. Centralized strategies are ones in which the load data of all objects in an application are collected on one processor and a new mapping of objects to processors is calculated. In distributed load balancers, processors communicate the load data of their objects with only a subset of processors and objects are exchanged among these subset of processors. Hybrid load balancers mix the characteristics of centralized and distributed load balancers by trying to take a global decision without having all processors send their load data to one processors. However, till date most applications in the CHARM++ system have used centralized load balancers to scale to large numbers of processors. We concentrate primarily on centralized load balancers in this thesis since they are the common case. But the methods we develop are also applicable to hierarchical load balancers such as those development in Zheng's thesis [53].

Figure 8.3 shows exactly how the different elements involved in load balancing interact within a processor. The load balancing framework is closely integrated with the CHARM++ runtime system and instruments all the messages processed and sent by all the objects on a processor. This load data for each object is stored in the *LBManager*(Load Balancer Manager) object on each processor. Whenever object α on processor A processes a message, the time taken to process that message is logged in the LBManager on processor A. Moreover, any messages sent out by object α as a result of processing a message are also logged, with the LBManager storing the size and destination of each message sent. As a result, the LBManager not only has an exact idea of the computation time utilized by each object but also the number of messages as well as the total size of messages sent to other objects. The user can however choose to not log communication data for a certain application.

The CHARM++ load balancing framework lets the application decide when exactly it wants to perform a load balancing step. All the objects in the application agree to do a load balancing step and as shown in Figure 8.3 signal their readiness by calling the *AtSync* method. On processor A these AtSync method calls get communicated to the CentralLB object (Centralized Load Balancer) on processor A. Once all the objects on processor A have called their AtSync methods, CentralLB asks the LBManager for the load data for all the objects on processor A. In the example shown in Figure 8.3, it is assumed that the user has chosen not to collect communication data. As a result, LBManager sends just the computation load for the different objects to CentralLB. The unit of computation load is normally cycles or seconds depending on the user choice. In the example objects α, β, χ have loads of 4, 2 and 3 seconds respectively. The CentralLB object collects this load data into a Statistics Message and sends it out to the designated *central* processor.

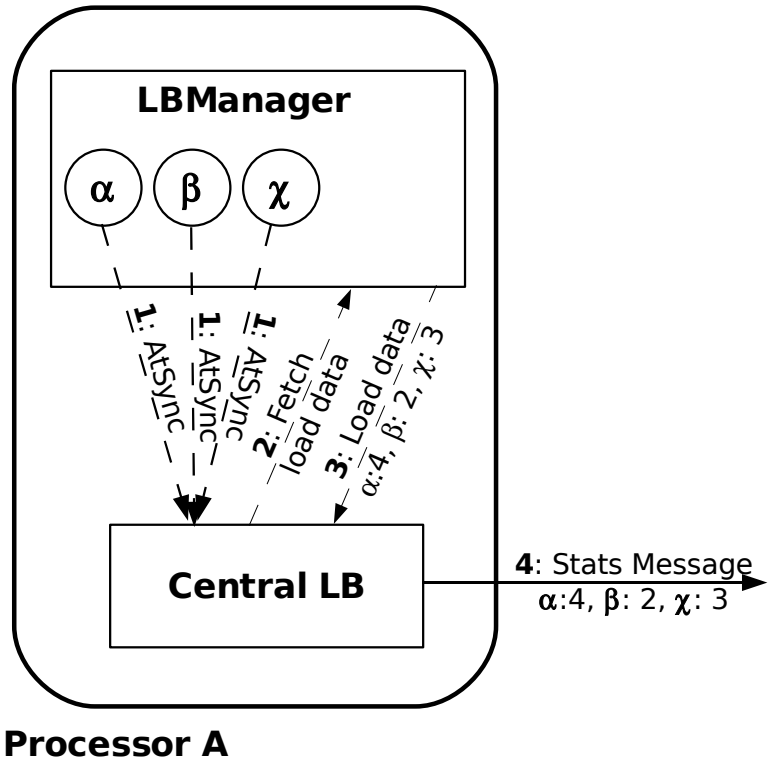


Figure 8.3: The interactions between Charm++ objects, LBManager and Central LB within a processor.

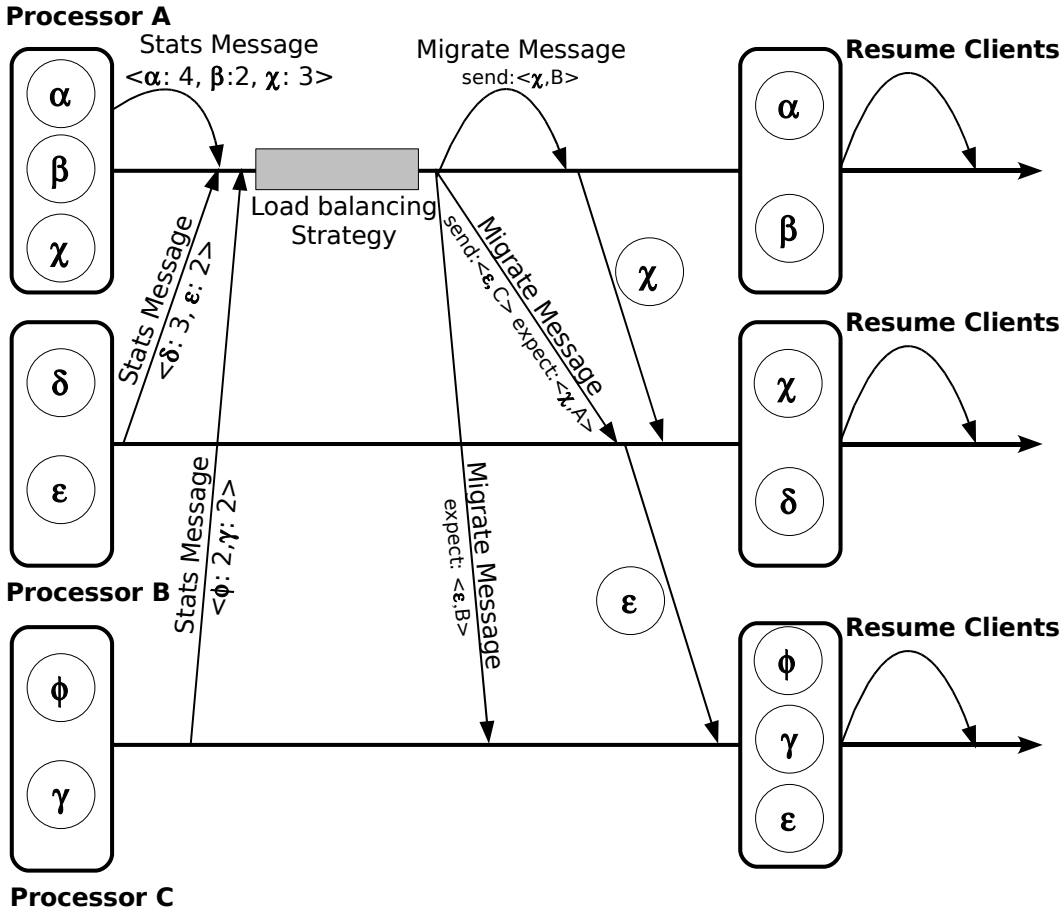


Figure 8.4: Messages exchanged during load balancing among processors

A processor is designated as the *central* processor for each load balancing step of centralized load balancers. The central processor can be changed from step to step. For the step shown in Figure 8.4, processor A is the central processor. Each processor sends its statistics message to the central processor. In the example, processors B and C send the statistics messages containing the computational load of their objects to processor A. Processor A collects all the data from the received statistics messages into one global load data structure.

Once processor A has received statistics messages from all processors, including itself, it invokes a centralized load balancing strategy on the global load data. The strategy calculates a new mapping of objects to processors based on this global load data. The user can choose exactly which load balancing strategy is invoked. The strategies range from those based on simple greedy algorithms on the computational load, to elaborate algorithms that map objects to machine topologies. After the load balancing strategy has calculated the new mapping, each processor is informed about the new mapping. Each processor is given a list of objects that it needs to send to other processors and another list of objects to expect from other processors. These lists are sent in a *Migrate Message* to each processor from the central processor. In Figure 8.4, processor B receives a Migrate message telling it to send object ϵ to processor C and expect object χ from processor A. Complementary Migrate messages are sent to processor A informing it that it should send object χ to processor B and to processor C telling it to expect object ϵ from processor B. After receiving a Migrate message, a processor packs any object it needs to send into a message and sends it. Once each processor has received all the objects it needs it calls the *Resume Clients* method to restart computation on every object existing on it.

8.3 Challenges in Merging Load Balancing and Message Logging

There are multiple challenges involved in trying to get the CHARM++ load balancing framework to work with message logging. We not only want to make sure that load balancing itself works correctly, but also that it does not break the fault tolerance provided by the message logging protocol. Moreover, the load balancing step itself needs to be fault tolerant. Any processor crashing in the middle of a load balancing step should not cause the application to hang or leave the global state of the computation in an inconsistent state. We divide the multifarious challenges into three broad categories:

- Effect of object migration on reliability
- Crashes during the load balancing step
- Interaction of load balancing and the fast restart protocols

We discuss these categories of challenges in the following subsections.

8.3.1 Reliability

The proof of our fault tolerance protocol for multiple faults in Chapter 5.3 hinges on the fact that for any message m, $Log(m)$ contains both $P(m.receiver)$ and $B(P(m.receiver))$. Since a processor and its buddy are assumed to never crash simultaneously $Log(m)$ does not ever become an empty set. If a processor goes down while its buddy is recovering from a crash of its own, we have an unrecoverable error. However, we showed in Chapter 5.2 that despite this potentially unrecoverable error, our protocol increases the reliability of a system by a few orders of magnitude. It gives an application a high probability of running successfully even in a high fault environment although it does not use any idealized

stable storage. The increase in reliability is provided by the fact that the objects on a processor are dependent solely on its buddy processor for their successful recovery.

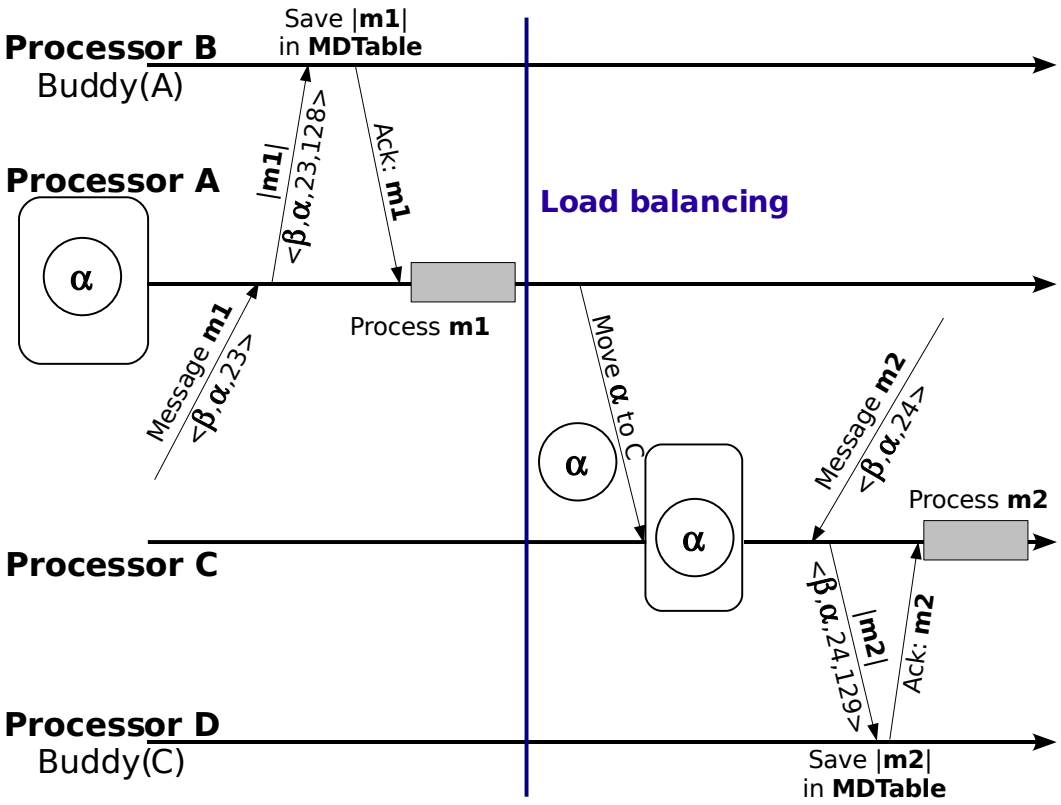


Figure 8.5: Illustrates the reliability created by migrating objects. Object α migrates from processor A to processor C

However, migrating objects during the execution of an application can potentially break this condition. Figure 8.5 shows this with the help of an example in which object α is migrating from processor A to processor C. It receives a message $m1$ from object β with a SN of 23. Object α assigns message $m1$ a TN of 128 and sends the meta-data of $m1$ to be logged on processor B, the buddy of processor A. Processor B stores the meta-data of message $m1$ in its MDTable and sends an acknowledgment back to processor A. After receiving the acknowledgment, object α processes message $m1$. After that, there is a load balancing step. For simplicity's sake, the load balancing step is represented as

a single event. As a result of that step, object α is moved from processor A to C. After that, object α processes another message ($m2$) from object β. Message $m2$ with a SN of 24 is given a TN of 129. The meta-data for message $m2$ is sent to the buddy of processor C, namely processor D, to record in its MDTable. Once the meta-data for message $m2$ has been added to the MDTable of processor D, an acknowledgment is sent back to processor C which then proceeds to process $m2$.

At this point, if processor C were to crash there are multiple problems with the recovery of object α. There is the basic problem of which processor has the checkpoint of object α and where should it be created. Processor D, the buddy of the crashed processor C, does not have its checkpoint. Therefore according to our current fault tolerance protocol, object α would not get created on processor C during its recovery. Moreover, object α has already been deleted on processor A. So, when processor C restarts, object *alpha* is neither on processor C or A and thus disappears from the system, preventing a full recovery. Of course, the checkpoint of α exists on some processor, the buddy of whatever processor it was on when it last checkpointed. Assuming that object α was on processor A when processor A last checkpointed, processor B might actually have a checkpoint of α. One could fetch object α's checkpoint along with its MDTable from processor B during the recovery of processor C. Although this would solve the problem of missing checkpoints it hurts the reliability of the system. Processor C is now dependent not only on processor D but also processor B for its recovery. If processor B were to crash, not only would the checkpoint of object α disappear but also the meta-data for some messages it processed. If processor C were to crash after processor B had crashed, we would end up with an unrecoverable error even though processor B is not a buddy of processor C or vice-versa.

The problem with the meta-data is potentially worse than that of the miss-

ing checkpoint. Let us assume that there have been a large number of load balancing steps since object α last checkpointed. So object α could have been temporarily located on a large number of processors. The last checkpoint of object α would exist on one processor and as long as that processor did not crash the checkpoint of α would be available during recovery. However, the meta-data for the messages processed by object α would be spread on the buddies of all the processors on which α was located since its last checkpoint. In the example in Figure 8.5, the meta-data for messages processed by α are found on both processors B and D, the buddies of processor A and C respectively. Thus, the recovery of object α is dependent on all the processors containing the meta-data of its messages being available. This increases the probability of failure since the crash of any of these processors would stop the recovery of object α. Moreover, it also invalidates the proof in Chapter 5.3 since $Log(m)$ is not necessarily $m.receiver$'s current processor and its buddy. Instead, $Log(m)$ is now the $m.receiver$'s current processor and the buddy of some processor object $m.receiver$ existed on earlier. For message $m1$ in Figure 8.5, $Log(m1)$ is $\{C, B\}$. Since, we do not make any assumption about the relation between the failures of processors C and B (they are not buddies) we invalidate our proof in Chapter 5.3.

8.3.2 Crashes During the Load Balancing Step

The load balancing step involves a large amount of data collection within a processor as shown in Figure 8.3 as well as messages between processors as in Figure 8.4. If a processor crashes, the current implementation of the load balancing framework will stall. This is because some of the communication between objects within one processor during load balancing uses function calls. Function calls are used because the load balancing framework is considered to be

a part of runtime system and uses these function calls instead of local messages as an optimization. However, this breaks a very basic assumption inherent in our protocol that objects interact with each other only through messages whose meta-data can be stored. As a result, during recovery we are not able to model all the state changes of objects on the recovery processor in the same sequence as before the crash. This brings the whole application to a grinding halt on the recovering processor.

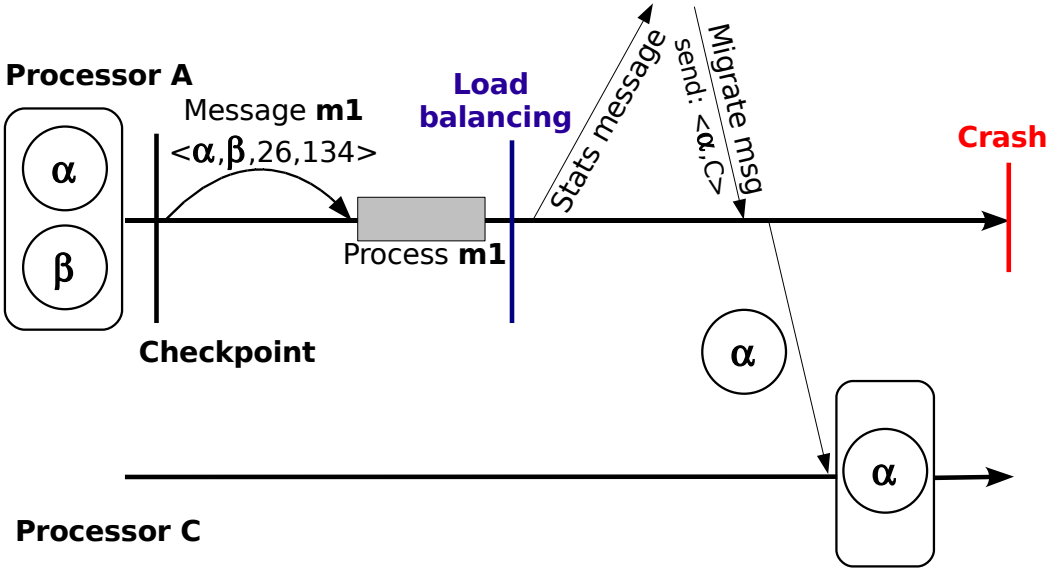


Figure 8.6: Object α is migrated from processor A to C during load balancing. Processor A crashes immediately after that.

Figure 8.6 illustrates another problem faced when there is a crash during a load balancing step. Initially objects α and β are both on processor A. Then processor A checkpoints, as a result of which the state of both objects α and β are saved. At some point after the checkpoint, object α sends message $m1$ to object β. Although Figure 8.6 does not show it for the sake of clarity, the meta-data of message $m1$ is logged on the buddy of processor A. After that the message $m1$ is processed. Since message $m1$ is exchanged between two objects on the same processor, it uses the local mode of the message logging protocol.

Therefore, once message $m1$ has been processed object α deletes its log of $m1$. This is followed by a load balancing step. Processor A receives a migrate message from the central processor asking it to send object α to processor C. Processor A packs up the state of object α and sends it to processor C. Processor C recreates object α from this message. Some time after this processor A crashes.

During the recovery of processor A, according to the current protocol, object α and β will both get recreated from their previous checkpoint. However object α already exists on processor C. If we were to recreate object α on processor A as well, we would end up with two copies of object α in the system. There are two possibilities at this point:

1. Leave the copy of object α on processor C and do not recreate α on processor A.

2. Pull down the copy of object α on processor C and recreate α on processor A from the checkpoint.

The first possibility seems more logical since, object α already exists on processor C and has probably made some progress since the last checkpoint. If we were to delete object α on processor C and recreate it on processor A from the checkpoint, we would seem to be not only undoing work unnecessarily but also increasing the work done during recovery. However, there is a problem with this solution. Object β needs message $m1$ from object α to complete its recovery. Object β has the meta-data for message $m1$, but it needs object α to either resend or regenerate the message itself. As mentioned earlier, since message $m1$ is a local message object α did not store its log once message $m1$ had been processed by object β. Therefore, object α can not resend message $m1$. So, object α has to regenerate message $m1$ and for that it needs to be

rolled back to the checkpoint. Of course, one could have solved this problem by storing the log of local messages as well. That would have let object α simply resend message $m1$. However, this would impose a massive memory overhead, since local messages are very common. This overhead would be imposed on the common case, just to improve the performance of a comparatively rarer event, that is recovery from a crash during a load balancing step. Therefore, we instead chose to go with the second possibility mentioned above. We delete object α on processor C and recreate it on object α on processor A from the previous checkpoint. This means that object α and β both go through their recovery together and all local messages between the two are regenerated. We describe the exact implementation in Chapter 8.4.

8.3.3 Load Balancing and Fast Recovery

The current load balancing protocol is designed with the assumption that it is the only source of object migrations in the system. If any object migrates into a processor, the load balancer always counts it as one of the objects expected during a load balancing. The fast recovery protocol of course breaks this assumption. After a processor crashes, the fast recovery protocol distributes its objects among other processors to speed up recovery. If on these recipient processors, the load balancer assumes that these migrations are due to itself, it can get confused with some processors ending up with more objects than expected and others less. Of course, this is a relatively easy problem to fix. Every object migration is marked as whether it is caused by load balancing or not. The load balancing framework only counts messages that have been marked as caused by load balancing.

However, the load balancing framework and the fast recovery protocol can still interfere with each other's functioning. Figures 8.7 and 8.8 show an example

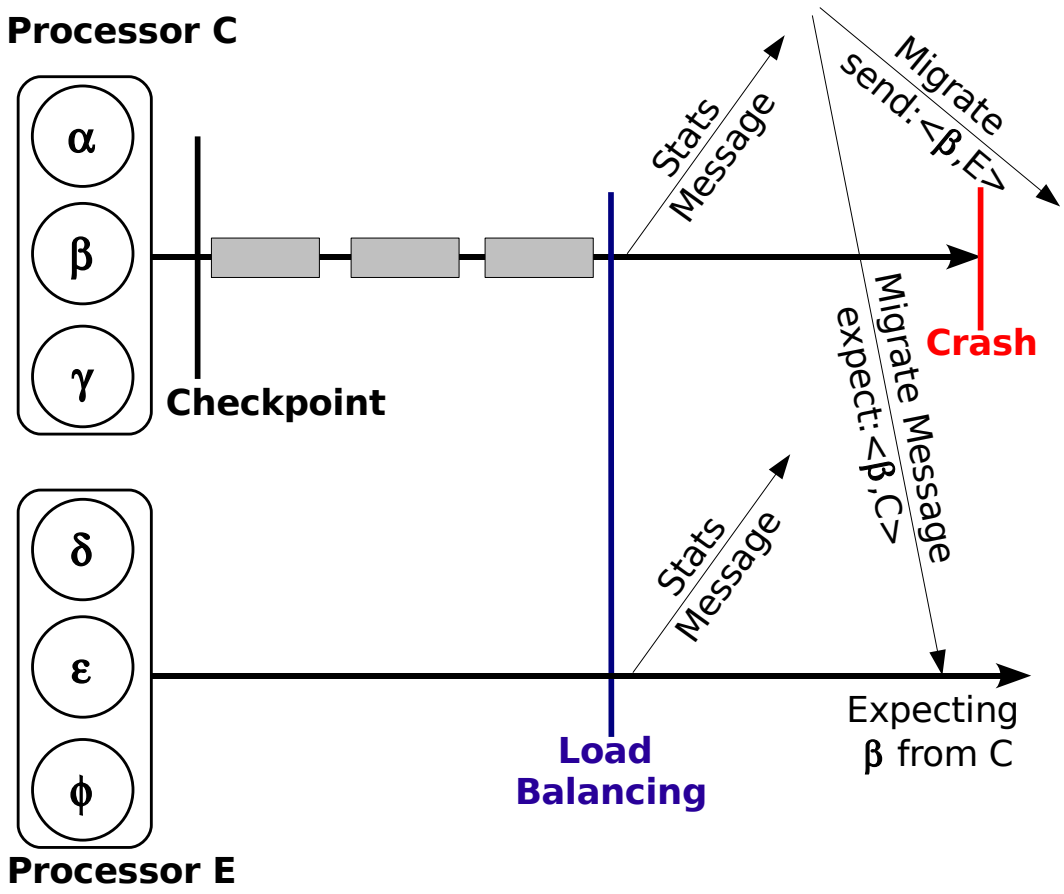


Figure 8.7: Processor C crashes during a load balancing step. However, processor E has already received a migrate message telling it to expect object β.

in which a crash occurs during load balancing. When we use the fast restart protocol to recover from the crash, it leads to processors expecting objects that they may never receive. Figure 8.7 lays out the situation just before the crash. Processor C has three objects, α, β and γ. Processor C checkpoints the states of these objects. Some time after that, load balancing begins. Processor A sends the statistics message for the objects on its processors. Once the central processor has calculated a mapping, it sends migrate messages to the different processors. The central processor sends a migrate message to processor E telling it to expect object β from processor C. Once processor C receives this message, it starts waiting for object β to arrive on it. However, before the migrate

message for processor C, telling it to send object β to processor E, can be received processor C crashes.

Processor C's crash triggers the fast recovery protocol. Figure 8.8 shows that the checkpoint of all the objects on processor C are fetched from its buddy. These objects are then distributed among other processors. Object α is retained on processor C. However, objects β and γ are sent to processors A and B respectively for their recovery. Sending the objects of course involves running the fast restart protocol described in Chapter 6, although we do not show it to avoid cluttering the diagram. Thus each object α, β and γ is recovered on a different processor. The recovery continues fine until the recovering objects reach the state they were in right before the load balancing step.

Once the recovering objects want to enter the load balancing step, there could be a few problems. Objects β and γ are no longer on processor C and their *AtSync* calls would get forwarded to the CentralLB object on their current processor and not on processor C as happened before the crash. However, this is basically the same problem discussed in Chapter 8.3.2. If the AtSync calls were to become local messages instead of being function calls, this problem would disappear. Therefore, this particular challenge is not a new one. However, once the objects enter the load balancing step other challenges crop up. When processor C receives the old migrate message, asking it to send object β to processor E, it can not do so since object β no longer exists on it. Similarly processor E has been waiting for object β to come from processor C, whereas processor C is in no state to send β. Therefore processor E's load balancing step would never end as it would keep waiting from object β which would never arrive.

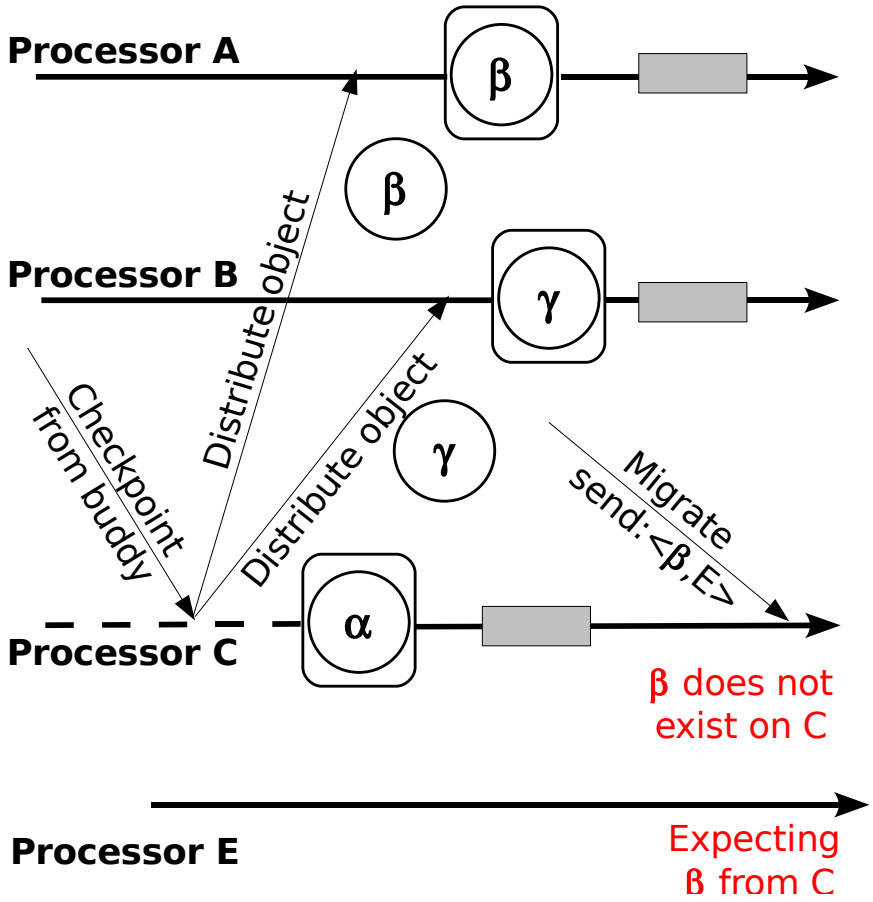


Figure 8.8: During the fast recovery of processor C, objects β and γ are distributed to processors A and B. As a result processor E is left waiting for object β

8.4 Fault Tolerant Load Balancing

We modified the existing load balancing framework as well as parts of the message logging protocol to get them to work with each other correctly and avoid the pitfalls described in the previous section. Most of the changes to the message logging protocol are limited to the recovery protocol. The changes in both parts of the run time system are of course closely intertwined. However, we describe them in separate subsections below to simplify the description.

8.4.1 Modified Load Balancing Step

The instrumentation and data collection parts of the load balancing framework are left untouched. However, the protocol of the load balancing step itself undergoes a major overhaul. The very first part, in which each object calls the *AtSync* method to notify its readiness for load balancing, is modified. The *AtSync* calls no longer result in function calls to the CentralLB object on the local processor. Instead, local messages are sent from the participating user objects to the CentralLB object. In case of a crash, these AtSync messages can be resent during recovery. This restores a basic assumption of our protocol that the state of any object is only affected by the messages it processes. Moreover, during fast recovery objects distributed among other processors can signal their readiness for load balancing to the CentralLB object on their original processor. Thus this also removes an obstacle in the path of getting load balancing and message logging to co-operate.

There is no change in the parts of the load balancing protocols that involve sending the statistics from a processor to a central processor, calculating a new mapping and sending migrate messages to the other processor. We can get by without modifying these messaging portions of the load balancing protocol, since the load balancing protocol is itself implemented using Charm++ objects (the CentralLB objects). These objects belonging to the runtime system also communicate via the message logging protocol. As a result, during recovery from a crash the CentralLB objects also resend necessary messages from their message logs. This greatly simplified the process of making the load balancing framework fault tolerant.

Although the statistics and migrate messages are Charm++ messages, the messages used to actually send objects from one processor to another can not

be CHARM++ messages. We can not use CHARM++ messages for sending objects since the objects themselves are the communicating entities. So, a CHARM++ message containing an object would break our idea of applying the PWD assumption to objects instead of processors. Thus, we deal with messages containing objects as a special case outside CHARM++, in which communication occurs purely between processors. These messages are assumed to not change the state of the object being communicated, just its location.

Figure 8.9 shows the different messages sent during a load balancing step. It leaves out the statistics collection part and starts from the migrate messages sent to different processors. Before load balancing, objects α and β are located on processors A and C respectively. The migrate message tells processor A to send object α to processor C and expect object β from processor C. Similarly processor C receives a migrate message that tells it to send object β to processor A and expect object α from A. At this point, we assume that the processors A and C both agree to send objects α and β to their destinations. Later, we discuss the case when a processor does not send an object on receiving a migrate message.

Processor A starts the migration of object α to C by sending a message to its buddy processor B. The message informs processor B of object α's intention to migrate to processor C. Processor B marks object α's checkpoint as *migrating* to processor C. Processor B then sends an acknowledgment message to processor A. After receiving the acknowledgment message, processor A sends object α to processor C. However, at this point processor A does not delete its copy of object α. It just marks it as being under migration. Similarly, processor C, informs its buddy processor D that it is migrating object β to processor A. Once processor D acknowledges this information, processor C sends object β to processor A.

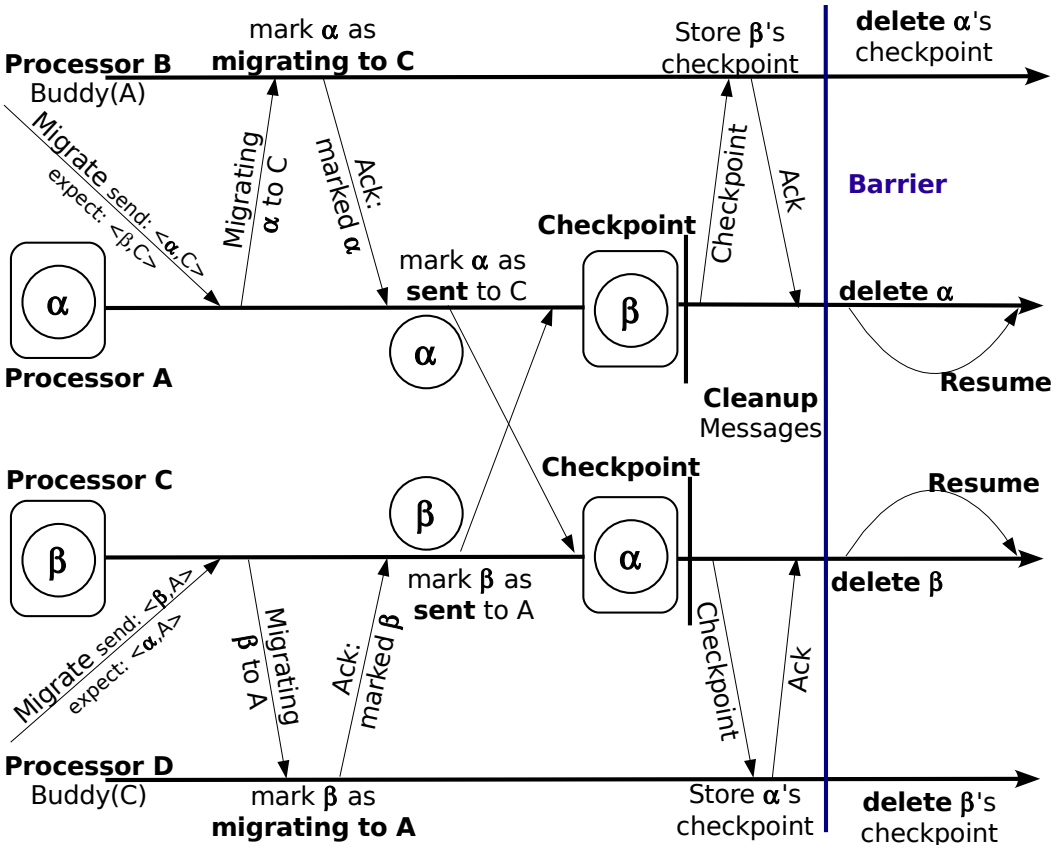


Figure 8.9: The messages involved in migrating an object during the load balancing step

Once a processor has received all the objects that it expects and has sent all the objects that it needs to, it takes its checkpoint. During the checkpoint, a processor does not include the state of the objects that are under migration. That means that when processor A checkpoints it includes the newly arrived object β, but not object α, which is under migration, in its checkpoint. As usual processor A's checkpoint is sent to its buddy processor B. At this point, processor B stores the checkpoint of object β. However, it does not yet delete the old checkpoint of object α. Processor B then sends an acknowledgment to processor A. Unlike the earlier case, processor A does not send out garbage collection messages when it receives the acknowledgment. Once all the processors have had their new checkpoints acknowledged they participate in a barrier.

The barrier marks the point in the system at which all the processors have had their checkpoints acked. This means that there is no chance that an object will have to be created from an older checkpoint. Therefore, at this point the old checkpoints and objects marked as migrating can be deleted. This means that processor A deletes the old copy of object α and processor B deletes the old copy of α's checkpoint marked as migrating. Moreover, objects can restart processing messages now. Garbage collection messages are also sent out at this point instead of being sent out when the checkpoint was acknowledged.

The modified load balancing protocol solves a number of problems mentioned in Chapter 8.3. The copy of object α is not deleted at its old location, processor A, as well as at A's buddy processor B, until its new location processor C has checkpointed. This means that through out the load balancing step, copies of object α are present on at least two processors such that one processor is a buddy of the other. Object α has copies on processors A and its buddy B until, it is confirmed that processor D, the buddy of processor C, has stored object α's checkpoint. This solves the reliability problem associated with checkpoints mentioned in Section 8.3.1. Of course, it also means that at some points there are more than two processors with a copy of object α. We describe in the next section how we modify the recovery protocol to deal with the situation. Since we always checkpoint during a load balancing step, object α does not require meta-data for any messages from before the load balancing step. Moreover, object α does not process any messages until the load balancing step is over. This means that object α's recovery is not dependent on meta-data stored on multiple different processors. Thus, we also solve the reliability problem associated with having meta-data spread over multiple processors as described in Section 8.3.1.

We now discuss the case where a processor might not send an object to

another processor after receiving a migrate message. Let us say processor E which contains objects γ, δ and ϵ crashes during the load balancing step. The crash happens right after processor E has received the migrate message. The migrate message asks processor C to send object δ to processor H but processor E crashes before any objects have actually been sent. During fast recovery of processor E, objects δ and ϵ are sent to processors F and G respectively. After recovery is nearly complete, processor E will receive the old migrate message again. However, processor E can not send object δ to processor H, since it does not have object δ. So, processor C sends processor H a message informing it that it will not get object δ and that it should stop waiting for it. The converse case that a processor is trying to send an object to the crashed processor is dealt similarly. The sending processor does not send the object to the recovering processor since it does not want to add to the load of the recovering processor. The sending processor simply lets the recovering processor know that it will not get that particular object and that it should not wait for the object.

8.4.2 Message Logging

The biggest modification to the message logging protocol happened in the recovery component. The recovery protocol had to be augmented to be able to deal with the various transient but confusing situations created by object migration during load balancing. Figure 8.10 shows an example that we use to illustrate the modifications. Processor C contains objects α and β. Processor C had saved the checkpoints of these object on its buddy, processor D, during the previous load balancing step. During the current load balancing step, it receives a migrate message telling it to send objects α and β to processors E and F respectively.

Processor C sends a message to its buddy processor D informing D that

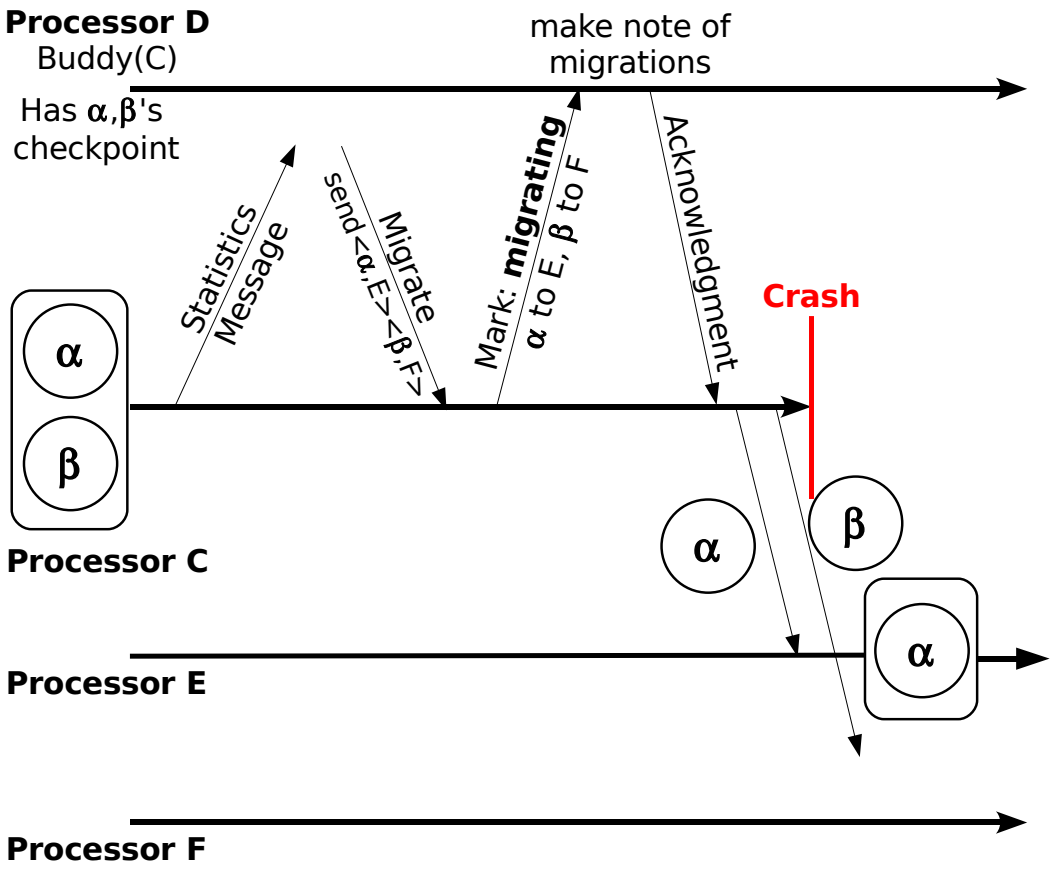


Figure 8.10: Processor C is sending objects α and β away, but crashes in the middle

object α is being sent to processor E and β to processor F. Processor D makes a note of these migrations in the object's checkpoint. After this, processor C starts sending the objects to their destination. It sends object α to processor E safely, but crashes before it can send out object β to processor F.

Figure 8.11 shows the steps in the recovery protocol in this case. As usual, a new processor C is started up that sends a request for its checkpoint to processor D. However, at this point the modified recovery protocol diverges from the old version. Processor D looks at the checkpoints of objects α and β and finds that they were being sent to processors E and F respectively. Processor D sends *verify* messages to processors E and F, asking each if the corresponding object

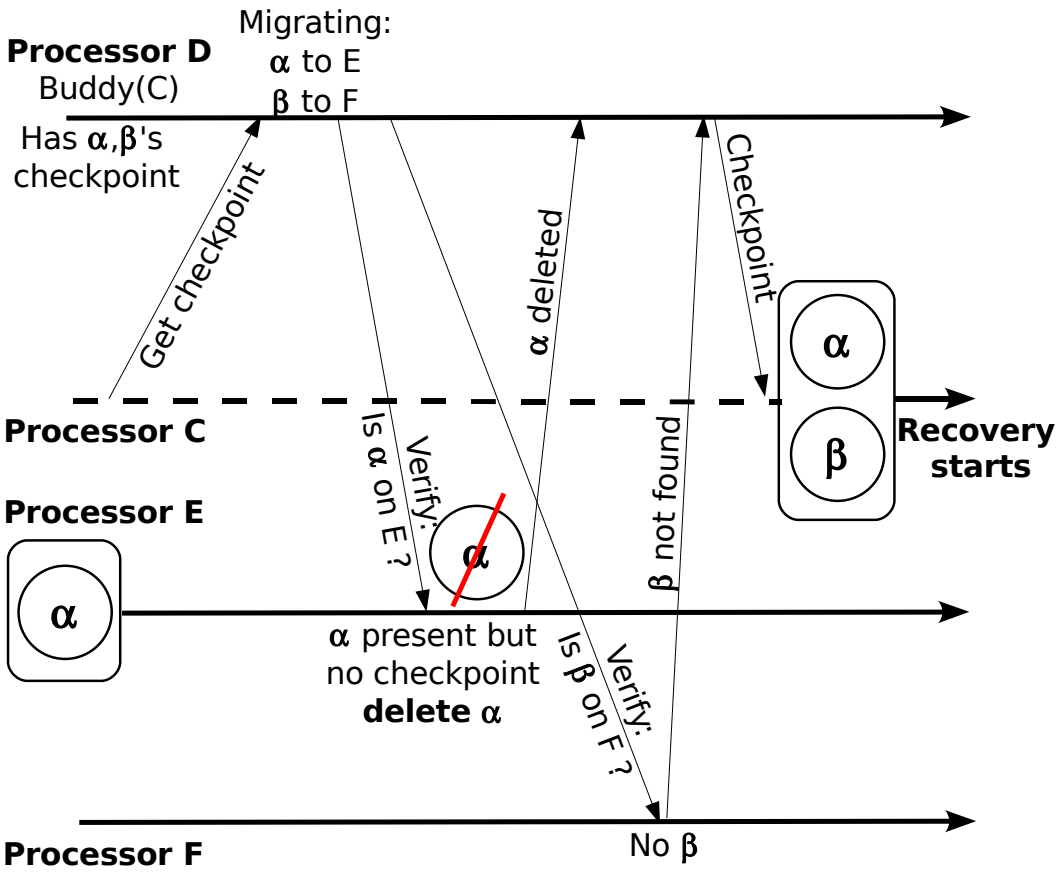


Figure 8.11: Recovery of processor C, after it crashed during the migration of objects α and β.

exists on it. In this case, processor E had successfully received object α from processor C before it crashed. So, processor E has the object that processor D is looking for. Moreover, it is found that processor E has not checkpointed yet during this load balancing step. Object α is simply deleted on processor E and processor D is told of the action. On the other hand, processor F does not have a copy of object β and it tells processor D the same. After hearing back from both processors E and F, processor D can be sure that it now has the only copies of objects α and β in the system. It sends their checkpoints to processor C and after that recovery can continue in the normal fashion.

Processor E might have already saved its checkpoint by the time the verify

message from processor D arrived. In that case, processor E does not delete its copy of object α. It simply tells processor E that it has object α and that it has also checkpointed. Now, if processor D receives a similar reply for object β, it realizes that there are newer versions of all the objects in the checkpoint of processor C. Moreover, these new versions themselves have checkpoints and are fault tolerant. So, processor D does not restore object α and β from their old checkpoints and sends processor C an empty checkpoint. Processor C then simply calls the checkpoint barrier for this interrupted load balancing step. On the other hand, if processor F had replied that it had no copy of object β or that it had deleted β, processor D would have to go through with the recovery of both objects α and β. Processor D has to rollback both processors to their previous checkpoint because of the reason explained in Chapter 8.3.2. So, processor D asks processor E to delete its and its buddy's copies of object α. After processor E and its buddy confirm deleting their copies of object α processor D can send the checkpoints of objects α and β to processor C.

Thus we extend both the load balancing framework and the message logging protocol to make them work together without compromising the reliability of the system. We provide an application with the performance benefits of load balancing while making sure that it can recover from a crash at any point during an execution.

Chapter 9

Experimental Evaluation of Protocol With Load Balancing

We now evaluate the performance of the combined message logging and load balancing protocol. This evaluation has two major parts: 1) showing that load balancing continues to improve performance even when combined with the message logging protocol 2) load balancing gets rid of the load imbalance created by the fast recovery protocol.

9.1 Load Balancing Without Faults

We want to make sure that load balancing continues to function correctly when used along with the message logging protocol. This means that load balancing should continue to correctly collect load information about different objects. Moreover, during a load balancing step it should use this collected load data to map objects to processors such that the load is distributed more or less equally among processors. We aim to assure that the extra messages and computation caused by the message logging protocol do not confuse the load balancing protocol.

We modified the AMPI 7-point 3D stencil application we used earlier. The modified version can increase the computational load of some virtual processors. The user can specify which virtual processors do more computation and also how much more. If the user does not specify higher computational load for any virtual processor, we get uniform load across all virtual processors just as in the

unmodified version.

We ran this modified stencil application with 512 virtual processors on 32 processors of the uranium system for this experiment. The base case was an execution without any overloaded virtual processors. For the runs with non-uniform loads we overloaded 10 of the 512 virtual processors. Each overloaded virtual processor does 8 times as much work as an unloaded one.

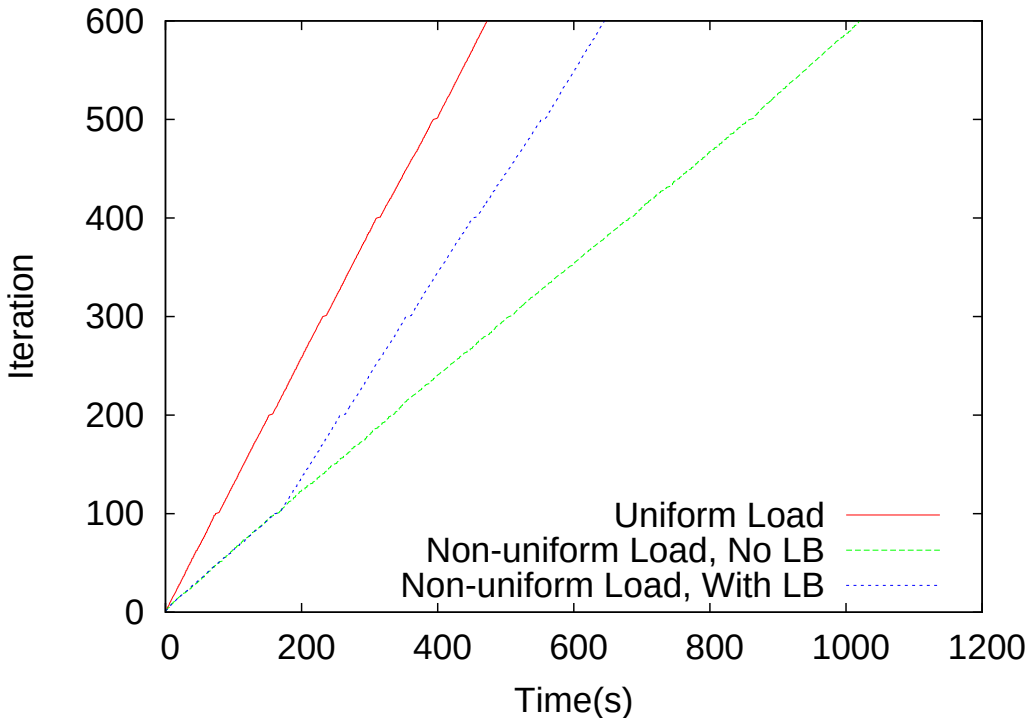


Figure 9.1: Progress of the AMPI 7-point 3D stencil application on 32 processors with uniform load, non-uniform load without load balancing and non-uniform load with load balancing. There are 512 virtual processors in all three cases.

Figure 9.1 shows the progress of the stencil application for all three cases: uniform load, non-uniform load without load balancing and non-uniform load with load balancing. We plot the iteration against the cumulative time since the start of the first iteration. A steeper line denotes faster progress. We checkpoint every 100 iterations for the runs without load balancing, and perform load balancing at the same frequency for the run with load balancing. These

checkpoint and load balancing steps show up as small flat lines in the progress curves. As expected the base case with uniform load makes the fastest progress. For the first 100 iterations, both the runs with non-uniform loads perform similarly. They make much slower progress compared to the uniform run, shown by their flatter progress lines. However, after the first load balancing step the performance of the non-uniform run improves markedly. It starts making much faster progress after load balancing. The slope of the progress curve becomes much steeper, though it is still flatter than the base case. This is to be expected, since we now are doing more total work per iteration (some objects are doing eight times more work). Figure 9.1 very clearly illustrates the massive difference in total execution time that load balancing makes. The non-uniform case with load balancing takes nearly 40% less time to run for 600 timesteps than the case without load balancing.

Type of Run	Average time per Iteration (s)	Increase over Uniform Load
Uniform Load	0.755	NA
Non-Uniform Load, Calculated Ideal	0.896	18.75%
Non-Uniform Load without LB	1.740	130.46%
Non-Uniform Load with LB	0.909	20.40%

Table 9.1: The measured average iteration time for the AMPI 7-point 3D stencil application on 32 processors with uniform load, non-uniform load without load balancing and non-uniform load with dynamic runtime load balancing. The calculated ideal average iteration time for the non-uniform load is also shown.

Table 9.1 compares the average time per iteration for the different runs. The average was calculated over the 99 iterations between 101 and 200. We choose this range because the first load balancing step happens between the 100^{th} and 101^{st} iterations. Apart from the three runs, we also show a calculated value for the best possible iteration time with non-uniform load.

We show how we derive the calculated iteration time. We say that each

virtual processor does 1 unit of work in the uniform load case. Since there are 512 virtual processors in all, the total amount of work for that case comes to 512 units. For the non-uniform case, 10 of the 512 virtual processors do 8 units of work each. Therefore the total amount of work in the non-uniform case is $502 + 10 * 8 = 582$ units. This means there is $\frac{582}{32} = 18.1875$ units of work per processor. Since the work within a virtual processor is not divisible, some processors will have 18 units of work and some 19 units of work. Therefore, the average iteration time will be close to $\frac{19}{16} * 0.755\ s = 0.896\ s$. As shown in Table 9.1 this average iteration time reflects a 18.75% increase over the base uniform load case. Moreover, a mapping that can possibly attain this performance is easy to calculate. Assign each overloaded virtual processor to a processor and then assign 8 normally loaded virtual processors each to the remaining processors. Distribute the remaining normally loaded virtual processors among the processors in a round robin fashion.

The initial mapping of the virtual processors to processors happens to be such that some processors contain more than one overloaded virtual processor. As a result, the average iteration time for the run with non-uniform load and no load balancing is very high, more than double that of the uniformly load base case. However, when the load balancer is used in the non-uniformly loaded case, the average iteration time goes down sharply. The average iteration time with the load balancer is very slightly (less than 3%) more than the ideal calculated runtime. This shows that the measurement based runtime load balancer continues to function effectively even when combined with the message logging protocol.

9.2 Load Balancing After Faults

We saw in Chapter 8 that by distributing objects from the recovering processor among other processors, the fast recovery protocol can set up a load balance problem for the future. Once the recovering objects have caught up with the rest of the computation, different processors end up with different loads. This load imbalance means that the performance of an application is worse after fast recovery than after basic recovery. We also saw how this performance loss can entirely wipe out the time saved during fast recovery. We decided to remove load imbalance by combining the CHARM++ runtime system's measurement based dynamic load balancing system with message logging. We now evaluate how effective load balancing is in getting rid of the load imbalance created by fast recovery. We evaluate two scenarios: 1) the load across all the virtual processors in a computation is uniform 2) the load distribution across virtual processors in non-uniform.

9.2.1 Uniform Load

We used the 3D 7point stencil application running on AMPI to evaluate the effect of load balancing after fast recovery from a crash. We ran the stencil application with 512 uniformly loaded virtual processors on 32 processors. We checkpoint every 100 iterations in the case of runs without load balancing. We perform load balancing every 100 iterations for the runs with load balancing. We introduce a fault close to the 185^{th} iteration for all the runs.

Figure 9.2 shows the progress of the stencil application with uniform load for three situations: basic restart without load balancing, fast restart without load balancing and fast restart with load balancing. The application shows near identical performance in all three cases for the first 100 iterations. This

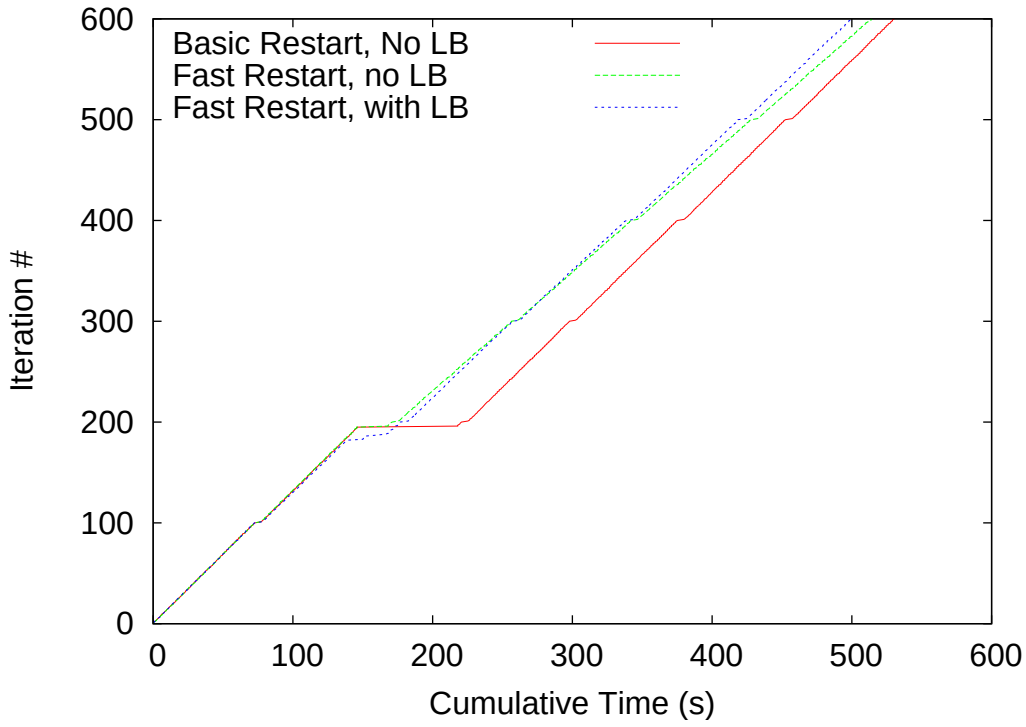


Figure 9.2: Progress of the AMPI 7-point 3D stencil application when faced with a fault under three different conditions: basic restart without load balancing, fast restart without load balancing, fast restart with load balancing. The experiments were run with 256 uniformly loaded virtual processors on 32 processors.

is to be expected because the initial mappings in all three runs are the same and load balancing has not occurred yet. Even after the first load balancing step at iteration 100, the performance remains similar. Table 9.2 shows the average iteration times in different phases of the application for these three cases as well as another one involving basic restart along with load balancing. The first column in Table 9.2 shows that there is very little difference in the average iteration time before the crash among the different runs. Since, the load on all virtual processors is uniform and the initial mapping distributed the virtual processors uniformly, performance does not change even after the first load balancing step.

All three runs suffer a processor crash and start recovery close to the 185^{th} iteration. This shows up as a flat stretch in the progress curves of all three runs. The length of the stretch represents the time taken in recovering from the crash. Once recovery is complete, the application can start making progress again. The basic restart run takes the longest to recover from the crash, about three times the fast restart runs. As a result, by the time the basic restart protocol finishes recovery, the fast restart runs have already made a significant amount of progress.

Type of Run	Average Iteration Time(s)		
	Before crash	Between recovery and next LB or checkpoint	After the next LB or checkpoint
Basic Restart, No LB	.734	.732	.738
Basic Restart, With LB	.732	.733	.740
Fast Restart, No LB	.725	.817	.820
Fast Restart, With LB	.723	.818	.748

Table 9.2: Compares the performance of the 3D stencil application during three different phases of the run for four different cases: before crash, after the recovery from the crash but before the next checkpoint/load balancing step and after the checkpoint / load balancing step following recovery

However, as seen earlier as well, the performance after recovery of the fast restart protocol without load balancing is significantly worse than that of the basic protocol. The second column in Table 9.2 shows that after recovery both the runs using fast restart protocol have a significantly higher average iteration time than the runs using basic recovery. The run that uses fast recovery without load balancing continues to have a higher average iteration time through out the rest of the application. We can see this in Figure 9.2, where the green line representing fast restart without load balancing has a lower slope after recovery than the basic restart. At some point, beyond the range shown, the green line would be crossed by the red line. That would represent the point when the load

imbalance created by fast restart wipes out the benefits of fast restart.

On the other hand, following a subsequent load balancing step the average iteration time for the run using fast recovery goes down sharply. This is borne out by the third column in Table 9.2. Similarly in Figure 9.2, the blue line representing fast recovery with load balancing has a much higher slope than the green line. The blue line remains more or less parallel with the red line showing that, when used with load balancing, fast recovery can maintain its advantage over basic restart. A small loss in performance for the blue line is caused by the fact that it takes a little longer to do every load balancing step than the red line takes for checkpointing. An user can avoid this repeated overhead by not doing load balancing steps unless there has been a crash during the previous phase.

We can fully exploit the advantages of the fast restart protocol only by combining it with load balancing. The recovery process is shorter than the basic restart protocol, while still preserving performance after recovery. Thus, load balancing combined with fast restart protocol provides better performance in the presence of faults, even for a uniformly loaded application.

9.2.2 Non-uniform Load

We now examine the effectiveness of combining load balancing and our fast recovery protocol for an application with non-uniform load. We use the same modified 7 point 3D stencil application as in Section 9.1 with the exact same initial configuration. There are 10 overloaded virtual processors among the 512 virtual processors and their initial mapping to physical processors is also the same as above. The test was run on 32 physical processors on uranium with either load balancing or checkpoints being performed every 100 iterations. We introduce a fault close to the 170^{th} iteration.

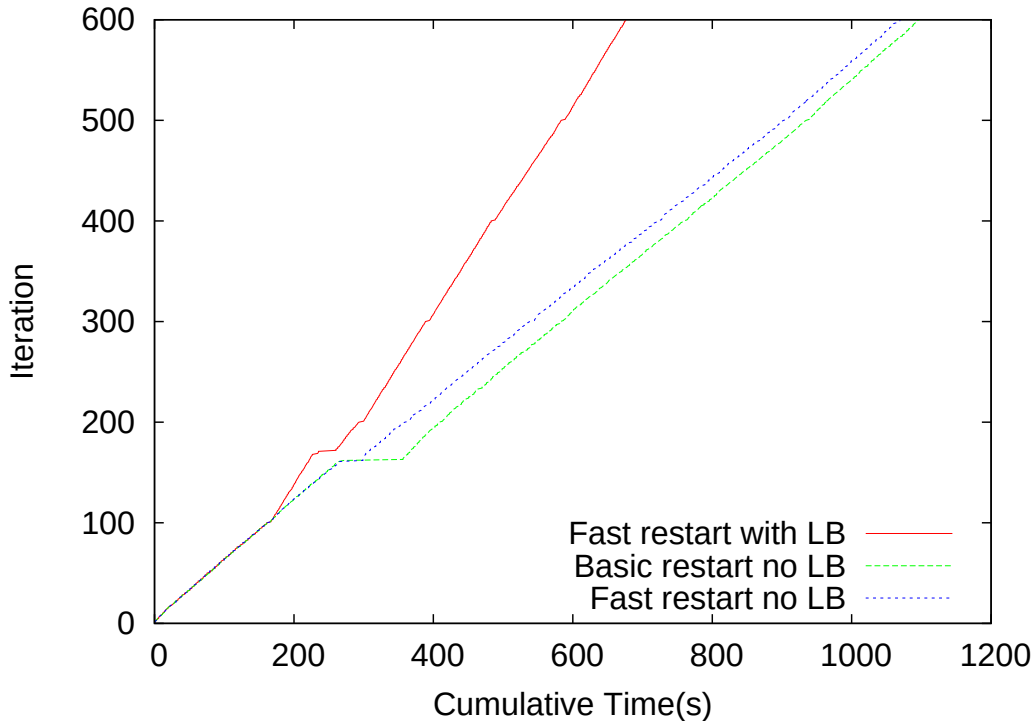


Figure 9.3: Progress of the 7 point 3D AMPI stencil application with 512 non-uniformly loaded virtual processors on 32 processors of uranium under three different conditions: fast restart with and without load balancing and basic restart without load balancing.

Figure 9.3 compares the progress of the non-uniformly loaded application under three conditions: fast recovery with load balancing, basic restart without load balancing and fast recovery without load balancing. As expected the performance of all three runs till the first load balancing/ checkpoint step is very similar. After the load balancing step the red line starts making much faster progress than the other two runs. The second data column in Table 9.3 shows that load balancing improves the average iteration time significantly for this non-uniformly loaded application. The runs in Figure 9.3 correspond to the top three rows in Table 9.3.

When a fault occurs, the fast restart protocols recover much faster than the basic restart protocol. The basic restart protocol without load balancing takes

more than double the time of either of the runs using the fast restart protocol. The first data column in Table 9.3 shows the recovery time for the different runs. It is interesting to note that the recovery time for the fast recovery protocol is shorter for the run with load balancing than the one without. This happens because load balancing at step 100 redistributes the overloaded virtual processors among all the processors. In the run with load balancing, the crashed processor does not happen to have any of the overloaded virtual processors. As a result, the recovery is not dominated by the time taken to re-execute the computation on a overloaded processor, unlike in the case of the run without load balancing.

After the recovery, the green line (basic restart, no load balancing) continues to make progress at the same rate as before. The runs employing the fast restart protocol suffer from the post-recovery load imbalance problem discussed earlier. The third data column in Table 9.3 illustrates this load imbalance and the performance impact it has. In fact, the run using the fast restart protocol without load balancing worsens the already bad load imbalance in this non-uniformly loaded application. The fast restart protocol with load balancing also takes a hit to its performance, but the average iteration time is still much lower than the other two runs in Figure 9.3, since its load was well balanced before the recovery. This performance hit shows up on the red progress curve in Figure 9.3 as a flatter stretch between iterations 180 and 200. However, after load balancing at step 200 the performance of the red line improves back to its pre-crash level. The run with fast recovery as well as load balancing makes much faster progress than the other two runs in Figure 9.3. It takes 35% less time to finish 600 iterations as the other two runs. This shows that using the fast restart protocol along with load balancing for a non-uniformly loaded application has many advantages compared to using either the basic or

fast restart protocol without load balancing.

Type of Run	Recovery Time (s)	Average Iteration Time(s)		
		100 to Crash	Crash to 200	200 to 300
Basic Restart No LB	89.727	1.668	1.654	1.702
Fast Restart No LB	33.206	1.660	1.721	1.768
Fast Restart With LB	25.116	.906	1.16	.901
Basic Restart With LB	48.114	.920	.915	.917
No crash With LB	NA	.902	.899	.916

Table 9.3: Compares the performance of the 3D stencil application for four different cases during three different phases of the run: before crash (iteration 100 to crash), after the recovery from the crash but before the next checkpoint/load balancing step (crash to iteration 200) and after the checkpoint / load balancing step following recovery (iteration 200 to 300).

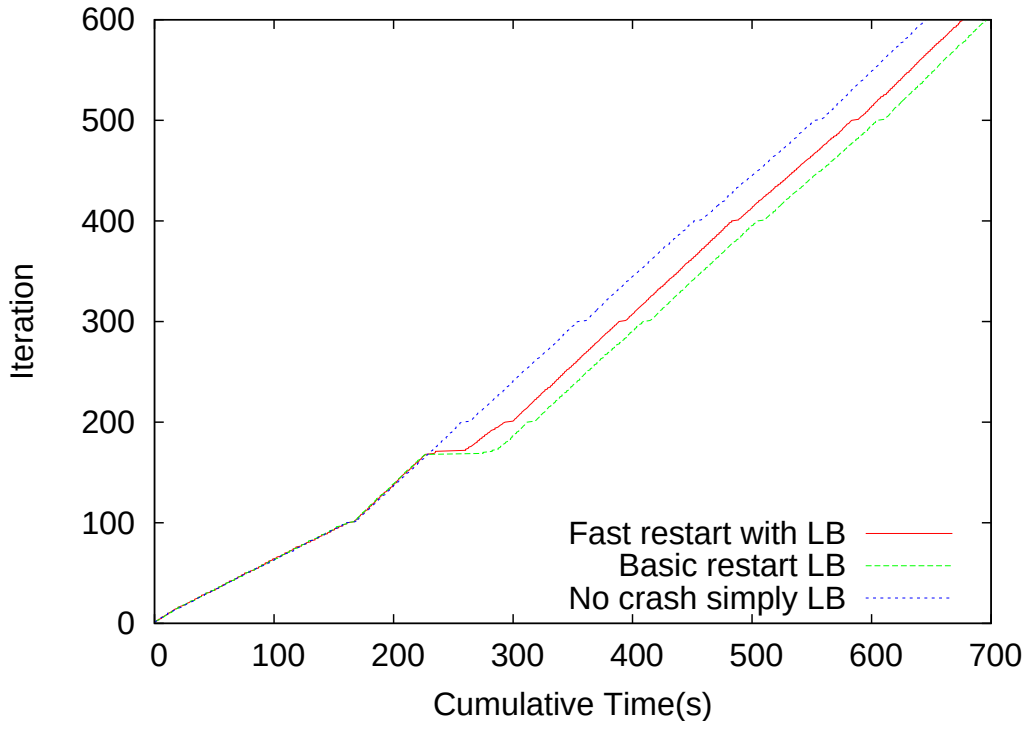


Figure 9.4: Progress of the 7 point 3D AMPI stencil application with 512 non-uniformly loaded virtual processors on 32 processors of uranium under three different conditions: fast restart with and without load balancing and basic restart without load balancing.

Figure 9.4 compares the progress of the run using fast restart protocol along with load balancing to two other runs: a run employing the basic restart protocol in tandem with load balancing and a run that suffers no crash but uses load balancing. The last run is used as a base case to compare the performance of the other runs with. These runs correspond to the last three rows in Table 9.3. Figure 9.4 and Table 9.3 show that the performance of the three runs is very similar in all phases of the application except in the vicinity of the restart protocol. The fast restart protocol with load balancing recovers in about half the time as the basic restart protocol with load balancing. It is interesting to note that the basic restart protocol takes much less time to complete recovery when it is using load balancing than compared to when it is not. This happens because the load balancing step at iteration 100 had reduced the load on the processor that crashes. Therefore, less work needs to be done during recovery of the crashed processor in the case of the run with load balancing. However, even with load balancing basic restart takes longer to recover from a crash than fast restart with or without load balancing.

As we saw earlier, the fast restart protocol suffers a performance penalty in the iterations immediately following a recovery. The basic restart protocol when used with load balancing performs better during this part of the computation since it does not disturb the load distribution among the processors. As a result, in between the iterations 180 and 200 the red line has a flatter slope than the green line in Figure 9.4. However, in this application the overhead is small enough that the fast restart protocol retains most of its advantage over the basic restart protocol before hitting the next load balancing step. After the load balancing step, the performance of all three runs is very similar. Therefore, the fast restart protocol used along with load balancing has a lower overall execution time than the basic restart protocol in this example. However, there

is a possibility that if the load balancing step after a crash were to happen a long time after the recovery, the slower performance of a run using fast restart protocol during that phase might again wipe out the advantage gained by doing fast recovery. We can avoid this problem by getting the runtime system to trigger a load balancing step immediately after recovery is complete. This can be investigated in greater detail in the future.

9.3 Comparing Performance With a Checkpoint Based Protocol

We want to compare the performance of our fault tolerance protocol with that of an existing checkpoint based protocol. The CHARM++ run time system provides us with two options: a disk based checkpointing protocol [30] and a double in-memory checkpoint protocol [54]. During a run, the disk based checkpoint protocol periodically stores the checkpoints of CHARM++ objects on the parallel file system of a machine. If a processor crashes during the run, the whole job is terminated. Later, the user can resubmit the job and have the execution restart from the last checkpoints saved on the parallel file system. The user can restart the execution on a different number of processors than the original run. The in-memory double checkpoint protocol also uses the idea of a *buddy* processor. A processor stores the checkpoint of objects on it in the memory of a buddy processor as well as its own memory. If a processor crashes, the whole execution is not terminated. Instead a new process is started on an extra processor. The objects on the restarted processor are recreated from their checkpoints on the buddy of the crashed processors. All other objects on all other processors are recreated from their previous checkpoints on the same processor. The recovery proceeds once all objects in the execution have been

recreated from their previous checkpoint. The in-memory protocol also has a mode which does not require an extra processor when a processor crashes.

The disk based scheme did not seem appropriate for a comparison with our message logging based protocol because it stores its checkpoints on the parallel file system and also needs a job to be resubmitted for an execution to recover from a fault. Storing a checkpoint on the parallel file system is bound to be much slower than storing it in the memory of another processor. This would make the comparison between the disk based protocol and our fast recovery protocol unfair. Therefore, we decided to compare our protocol with the in-memory double checkpoint based protocol. Both the schemes store their checkpoints in the memory of a buddy processor and can use a pool of extra processors to continue execution if a processor were to crash.

We used a 2 dimensional 5-point stencil application written in Charm++ as a benchmark to compare the performance of the two protocols. Figure 9.5 shows the progress of the application using the two different protocols, when run with 1024 virtual processors on 128 processors of Abe. The fast recovery protocol is used along with load balancing. Load balancing is performed every 200 iterations. The checkpoint based protocol also takes checkpoints every 200 iterations, but does not perform any load balancing. Since all the objects are uniformly loaded and the initial mapping is well balanced load balancing is not really needed for this application. The fast restart protocol uses load balancing to get rid of the load imbalance problem created by the fast recovery. We insert two faults into each run, one after about 370 iterations, the other around the 900^{th} iteration.

Figure 9.5 shows that in the first 370 or so iterations before the first crash, the application makes faster progress when using the in-memory checkpoint based protocol rather than the fast restart protocol. This is expected since mes-

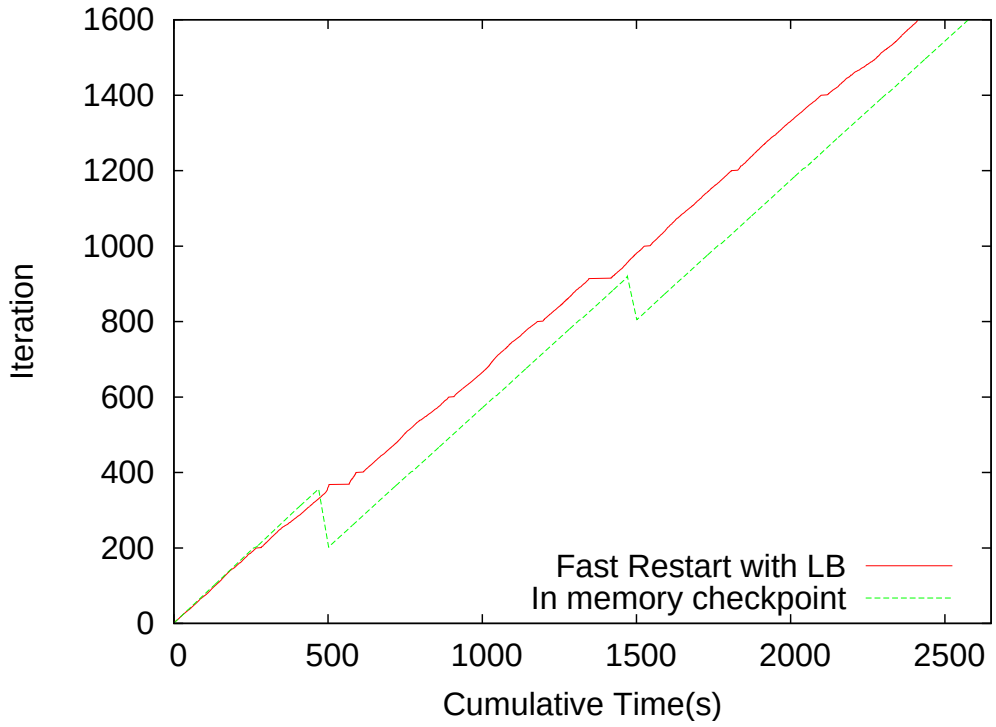


Figure 9.5: Compare the progress of a 128 processor 2D 5 point CHARM++ stencil program when run using the fast restart message logging protocol along with load balancing against while using an in-memory checkpoint based protocol.

sage logging has a performance penalty. Moreover, the combined load balancing and checkpointing step performed by the fast restart protocol takes longer than the simple checkpoint step of the in-memory checkpoint protocol.

After the first fault, the fast recovery protocol recreates objects from the crashed processor and distributes them among other processors. The in-memory checkpoint protocol on the other hand rolls all CHARM++ objects back to their previous checkpoints. This difference means that a crash and subsequent recovery show up differently in the progress curve for the two protocols. For the fast recovery protocol, recovery is marked by a horizontal stretch at which the rest of the application waits for the recovering objects to catch up. In the case of the in-memory checkpoint, there is a nearly vertical fall to the previous

checkpoint since all objects are rolled back to their previous checkpoints. Then, re-execution starts on all the processors and can be considered complete when the progress curve passes the iteration at which it originally crashed. The fast recovery protocol takes about 65 *seconds* to recover from the crash. This includes not just re-executing the work of the recovering objects but also re-spawning a new processor, fetching its checkpoint and distributing the objects among other processors. The in-memory checkpoint protocol takes 238 *seconds* to complete its recovery (including re-execution of the lost steps). Thus as expected, the checkpoint based method takes much longer to recover from a crash than the fast recovery protocol. The effect of the faster recovery shows up in the gap between the progress curves of the two protocols.

The second fault happens a little after the 900^{th} iteration. As for the first crash, the fast restart protocol recovers much faster from a crash than the checkpoint based protocol. The fast protocol takes 71 *seconds* to complete recovery and start making progress again. While using the in-memory checkpoint protocol, it takes the application 186 *seconds* to regain the state it had before the crash. Thus, again the fast restart protocol saves time compared to the checkpoint protocol while recovering from a crash. The fast restart protocol finishes 1600 iterations 164 *seconds* before the checkpoint protocol. Thus, we see an example in which the fast restart protocol has a lower overall execution time when compared to the in-memory checkpoint protocol.

Figure 9.6 shows a similar comparison between the fast restart protocol and the in-memory checkpoint protocol for the 2 D stencil application running on 512 processors on Abe. This example uses a larger data-set than the previous one and has 4096 CHARM++ objects in all. The fast restart protocol load balances every 200 timesteps. The checkpoint protocol takes checkpoints at the same frequency. In the initial stages of the application, the application

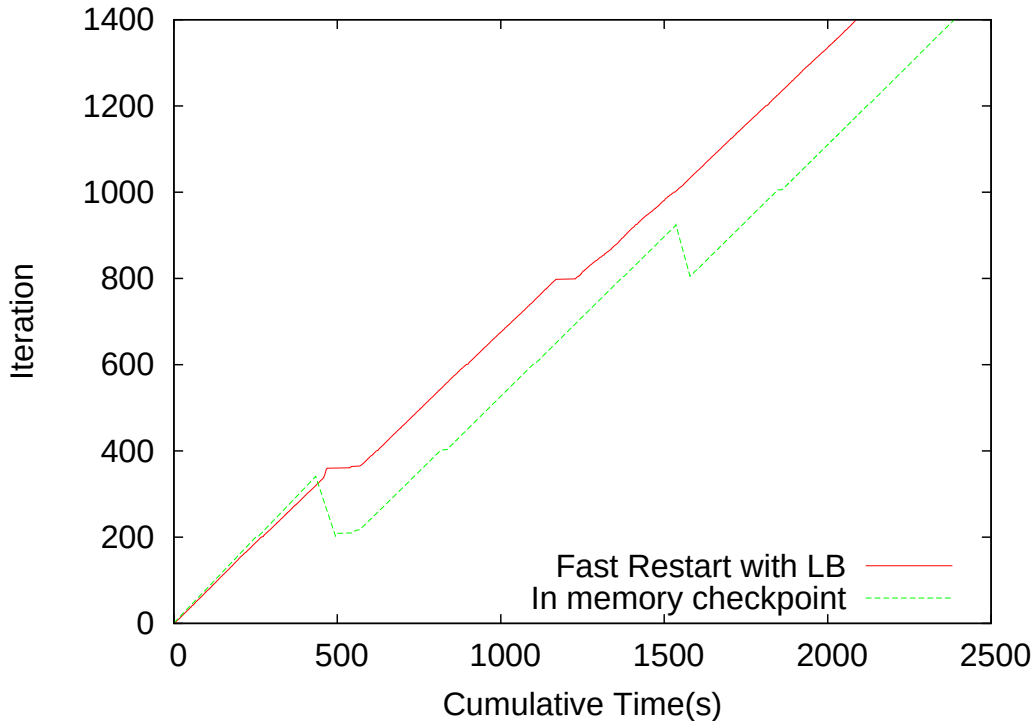


Figure 9.6: Compare the progress of a 512 processor 2D 5 point CHARM++ stencil program when run using the fast restart message logging protocol along with load balancing against while using an in-memory checkpoint based protocol.

makes faster progress while using the checkpoint protocol than the fast restart protocol. The first crash takes place around the 340^{th} iteration. While using the fast restart protocol the application takes 68.6 *seconds* to recover from the crash and to start making progress again. However, when used with the checkpoint protocol the application takes 280 *seconds* to reach the state it was in before the crash.

The second crash happens at slightly different times for the two runs. This happened because of the difficulty of triggering a crash at an exact point in the execution, particularly after an initial crash. In fact, for the fast restart protocol the crash happens in the middle of the load balancing step after iteration 800. The objects on the crashed processors are rolled back to their

previous checkpoints, that is the ones taken at the end of the load balancing step after the 600^{th} iteration. The fast restart protocol manages to recover from the crash during the load balancing step and takes 65 *seconds* for the recovery. The in-memory checkpoint protocol crashes around the 920^{th} iteration. All objects are rolled back to the checkpoints taken after the 800^{th} iteration. The checkpoint protocol in fact has to redo fewer iterations than the fast recovery protocol. Still, the checkpoint protocol takes 204 *seconds*, much longer than the fast restart protocol, to complete recovery.

The crash during load balancing in the fast restart case illustrates a side effect of our fault tolerant load balancing protocol. After the crash, the objects on the recovering processor are distributed among other processors. This includes any objects that might have been destined to move to other processors from the recovering processor according to the ongoing load balancing step. At the same time, objects on other processors destined for the recovering processor are no longer sent to it. Therefore, at the end of the recovery and the ongoing load balancing step there might be a load imbalance among the different processors. This is in fact the case since the average iteration time in the timesteps between 800 and 1000 is 1.54 *seconds* instead of 1.36 *seconds* during other fault free phases of the application. This 13.24% increase corresponds closely to the increase in iteration time that would be caused if some processors had 9 CHARM++ objects instead of 8. Therefore, during this time period, the gap between the run using fast recovery and the run with in-memory checkpoint closes somewhat. However, there still remains a large gap between the progress of the two runs with the fast restart protocol retaining a significant advantage over the checkpoint protocol. Moreover, after the next load balancing step at iteration 1000, the run using fast restart and load balancing redistributes the objects among the processors. This restores the load balance among the pro-

cessors in the run and removes the performance penalty imposed by the load imbalance created during fast recovery. So, at the end of 1400 iterations the execution time using the fast restart protocol along with load balancing is still 300 *seconds* ahead of the in-memory checkpoint protocol.

Chapter 10

Memory Overhead

Our protocol stores a significant amount of additional data in order to provide fast recovery from faults. Objects sending messages store these messages in their message log. Checkpoints of objects on a processor are stored in the memory of the buddy of that processor. Moreover, in order to deal with multiple faults message logs should be part of an object's state. In addition, there are protocol related data structures such as the TNTable, MDTable and SNTable. So, the fast recovery protocol increases the memory consumption of an application. In this section we measure the memory overhead imposed by our protocol on a simple application. We also look at methods of reducing the overhead.

We chose to investigate the memory costs of our protocol using a simple 2D stencil application. The configuration we used for the experiment involved a domain containing 268 *million* elements arranged in a 2D square gird with sides of length 16384. This domain is divided among 256 virtual processors with 16 virtual processors along each side of the domain. The application was run on 32 processors. It was run for 1000 iterations with checkpoints or load balancing happening every 200 iterations. The Charm++ run time system traps memory allocation and deallocation calls. This lets us keep track of the amount of memory being used on a processor by an application at any point of time. At the end of a run, we can also find out the maximum amount of memory used during the run on any processor. We decide to use this high water mark as a measure of the memory consumption of an application with and without the

fast restart protocol. This is a valid but somewhat conservative metric since the average memory consumption can be much lower than the maximum. On the other hand, the maximum memory consumption determines whether an application will actually run on a machine without swapping or even crashing due to a lack of physical memory.

We first ran the 2D stencil application in the described configuration with an unmodified version of CHARM++. We ran the application without performing load balancing every 200 timesteps. The maximum memory consumed on any processor during the run comes to 69.9 $MBytes$. This seems appropriate since each CHARM++ object has close to 8.5 $MBytes$ of data and there are 8 objects per processor.

Next, we ran the 2D stencil application with the fast recovery protocol but without load balancing. We performed a checkpoint every 200 iterations. The maximum memory used over all processors was 487.7 $MBytes$. This seemed surprisingly large. We started investigating this massive increase in memory consumption by looking at the checkpoint size. We found that checkpoint sizes for each processor went up to 121 $MBytes$. Of this, the application data accounted for 68 $MBytes$. The message logs at 52 $MBytes$ accounted for the bulk of the rest of the checkpoint. Thus, just the size of the checkpoint was not sufficient to explain the much higher memory consumption.

We decided to look into the memory consumption at different points of the checkpoint protocol. We found that the maximum memory consumed during a checkpoint protocol varied between different checkpoint steps. Table 10.1 compares the current memory consumption at different stages of the checkpoint protocol for two checkpoint steps on the same processor (processor 28): 1) during which memory consumption reaches the peak 2) during which memory consumption is significantly lower than the peak. At the beginning of the

Phase of the checkpoint protocol	Memory usage(MB)	
	Peak-usage	Non-peak
After sizing up the checkpoint	247.3	247.1
After allocating checkpoint buffer	367.5	367.2
After sending the checkpoint	487.7	367.2
Before storing received checkpoint	487.7	367.3
After storing received checkpoint	367.5	247.2
After receiving the checkpoint ack	247.3	247.2

Table 10.1: Compares the current memory consumption at different points in the checkpoint protocol for two different checkpoint steps on the same processor. The peak memory usage during the first step is also the maximum memory consumption during the run. The second run shows memory consumption for a checkpoint step whose maximum usage is less than the peak.

checkpoint, both steps have similar memory consumption, a little more than 247 $MBytes$. This memory consumption is expected since processor 28 contains its objects along with the checkpoint of the processor whose buddy it is (processor 3). As expected the memory consumption in both steps increases when the message to contain the next checkpoint is allocated. After that, the checkpoint message is filled up and sent to the buddy of processor 28. However, the message is not freed until the underlying interconnect decides that all of it has been received safely on the buddy. So, even after the message has been sent the memory consumption does not fall in the case of the non-peak checkpoint step.

The peak checkpoint step, on the other hand, shows an increase in memory consumption at this point. This happens because processor 28 receives the checkpoint message from processor 3. Thus, it ends up receiving that checkpoint message from processor 3 before it has sent out and freed up its own checkpoint message. Since it already contains its own objects along with a pre-existing copy of processor 3's checkpoint, it ends up with a total memory consumption about 4 times the checkpoint size. The checkpoint size of 120 $MBytes$ and current

memory consumption of 487.7 $MBytes$ adheres very closely to this scenario.

The next step of the checkpoint protocol has processor 28 processing the checkpoint message that it receives from processor 3. In the case of the non-peak step, by the time processor 28 receives that message from processor 3, it has sent off its own checkpoint message safely to its buddy. Therefore, memory consumption does not change significantly between sending its checkpoint and beginning to process processor 3's checkpoint message. After processor 3's new checkpoint message has been stored, the old one is deleted. This brings down the memory consumption for both the peak and non-peak step. In the peak memory usage checkpoint step, processor 28 deletes its own checkpoint message at some point after this, when the interconnect layer decides that the checkpoint message has been received safely on the buddy. Therefore, by the time the checkpoint protocol finishes with the acknowledgement from the buddy, memory consumption in both steps is back to the pre-checkpoint level of 247 $MBytes$.

Therefore, the high memory overhead is caused by processor 28 having 3 checkpoint messages along with its objects in its memory at the same time. An evident solution we tried was to store the received checkpoint on local disk. This would reduce the maximum amount of data contained in memory at one point. It does reduce the maximum memory consumed on any processor during the whole run to 368.3 $MBytes$. However, this is still pretty high and moreover the time taken to write the checkpoint to disk is close to 10 $seconds$. Another problem is that many machines do not provide compute nodes with a local disk. So, although saving the checkpoint to disk does reduce memory consumption, it is not the ideal solution.

We approached the problem from another direction by noticing that half the memory consumption is caused by the checkpoint messages received from another processor. If we rule out using local disks, then we will always have two

such messages in memory: one for the previous checkpoint step and another for the current checkpoint step. Since we can not delete the old checkpoint without first storing the new one, we are bound to have two such checkpoint messages in memory at some point of the execution. Therefore, the only way to reduce the memory overhead caused by the received checkpoint messages is to reduce the size of the messages themselves. We tested this idea by compressing the checkpoint message sent by a processor to its buddy. We used the *compress2* function provided by the *zlib* library to compress the checkpoint message. We send this compressed checkpoint message to the buddy. The buddy stores the compressed message. If a processor crashes, it receives this compressed checkpoint message and uncompresses it before starting recovery. It must be noted that the large buffer for the checkpoint message is still allocated to create the checkpoint message.

The compression was very effective and was able to reduce the checkpoint message size from 120 $MBytes$ to less than 10 $MBytes$. The compression took about 2.1 *seconds* on the slow processors of uranium. However, it also reduced the time taken to transmit the checkpoint from a processor to its buddy. It brought down the maximum memory consumption to 254 $MBytes$. Thus, compressing the checkpoint message drastically reduced the memory overhead caused by storing these messages without a big increase in the time taken to checkpoint. We leave compression as an option for the user to choose if memory consumption becomes too large for the amount of memory available on a machine.

We carried out a similar experiment with load balancing enabled. We performed load balancing every 200 timesteps. We found that the peak memory consumption on any processor while using load balancing but not the fault tolerance protocol was 147 $MBytes$. Load balancing increases peak memory

consumption since on some processors objects migrate in before their objects have migrated out. This can double the memory consumption on a processor and accounts for the higher peak usage during load balancing even when not using the fault tolerance protocol. The fault tolerance protocol suffers from a similar effect. In fact, the fault tolerance protocol has a higher chance of having this problem since a processor does not delete an object that is migrating away until its receiving processor checkpoints. So, the fault tolerance protocol, when used with load balancing but without checkpoint compression, has a maximum memory consumption of 603 $MBytes$. Compressing the checkpoint messages reduces the peak memory usage to 396.6 $MBytes$. Thus, compressing the checkpoint messages helps reduce the memory consumption of the fault tolerance protocol even when used with load balancing.

The compression technique in fact opens up a further avenue for reducing the memory consumption during the fast recovery protocol. The current implementation calculates the size of the uncompressed checkpoint and then allocates a message of that size before packing the object states into that message. The compression happens only after this. This means that the maximum memory consumption is bound to be larger than the sum of the size of the objects themselves and the size of their uncompressed checkpoint state. So, we can reduce the maximum memory consumption by not allocating the checkpoint message for the uncompressed size. We could use the stream compression functionality of the zlib library to compress the object state while packing it. This would reduce the maximum memory consumed during checkpointing by removing the need for the large uncompressed buffer. Moreover, the same idea can be used while moving objects during load balancing. The objects themselves as well as the copies retained can be compressed to reduce the peak memory consumption. However, this involves changing the packing unpacking framework

in CHARM++ significantly. So, it is left as future work if the peak memory consumption becomes an even bigger constraint.

Chapter 11

Proactive Fault Tolerance

We have presented and evaluated a fault tolerance protocol for recovering quickly from processor crashes. In this chapter we attack the problem of fault tolerance from a different angle. Instead of waiting for faults to occur and then recovering from them, we proactively migrate the execution from processors where failure is imminent. We exploit the capability of runtime migration provided by object based virtualizaiton to evacuate such processors. We evacuate the execution of an application from these processors to the other processors involved in the same application. We do not require extra *spare* processors while evacuating processors that might crash shortly in the future. We modify the run-time system such that if the warned processors were to crash, the rest of the computation can continue unhindered. We would like to point out that this work, in its current state, is independent of the fault tolerance protocols discussed until now.

This approach requires that failures be predictable. We leverage the fact that current hardware devices contain various features supporting early fault prediction. As an example, most modern disk drives follow the SMART protocol [4], and provide indications of suspicious behavior like transient access errors, retries, etc. Similarly, motherboards contain temperature sensors, which can be accessed via interfaces like lm_sensor [2] and ACPI [29]. Meanwhile, many network drivers, like those for Myrinet interface cards [10], maintain statistics including packet loss and retransmission counts. In fact, the PAPI-4 toolkit pro-

vides information from ACPI temperature sensors and Myrinet counters [41], in addition to the traditional hardware performance counters.

Processor manufacturers, are building infrastructure to detect transient errors inside processor chips and notify the O.S. [8]. Furthermore, recent studies have demonstrated the feasibility of predicting the occurrence of faults in large-scale systems [45] and of using these predictions in system management strategies [43]. Hence, it is possible, under current technology, to act appropriately before a system fault becomes catastrophic to an application. We focus on handling warnings for imminent faults and not on the prediction of faults. For faults that are not predictable we can revert back to traditional fault recovery schemes, like checkpointing and message logging.

Our strategy is entirely software based and does not require any special hardware. However, it makes some reasonable assumptions about the system. The application is warned of an impending fault through a signal to the application process on the processor that is about to crash. The processor, memory and interconnect subsystems on a warned node continue to work correctly for some period of time after the warning. This gives us an opportunity to react to a warning and adapt the runtime system to survive a crash of that node. The application continues to run on the remaining processors, even if one processor crashes.

We decided on a set of requirements before setting out to design a solution. The time taken by the runtime system to change (*response time*), so that it can survive the processor's crash, should be minimized. Our strategy should not require the start up of a spare process on either a new processor or any of the existing ones. This eliminates the need to maintain a pool of extra processors in case of a crash, as well as the overhead associated with running two application processes on one processor, albeit one of the application processes has no user

data. When an application loses a processor due to a warning, we expect the application to slow down in proportion to the fraction of computing power lost. Our strategy should not require any change to the user code. We verify in Section 11.2 how well our protocol meets these specifications.

11.1 Fault Tolerance Strategy

We now describe our technique to migrate tasks from processors where failures are imminent. Our solution has three major parts. The first part migrates the Charm++ objects off the warned processor and ensures that point-to-point message delivery continues to function even after a crash. The second part deals with allowing collective operations to cope with the possibility of the loss of a processor. It also helps to ensure that the runtime system can balance the application load among the remaining processors after a crash. The third part migrates AMPI processes away from the warned processor. The three parts are interdependent, but for the sake of clarity we describe them separately.

11.1.1 Evacuation of Charm++ Objects

Each migratable object in Charm++ is identified by a globally unique index which is used by other objects to communicate with it. We use a scalable algorithm for point-to-point message delivery in the face of asynchronous object migration, as described in [39]. The system maps each object to a *home* processor, which always knows where that object can be reached. An object need not reside on its home processor. As an example (shown in Figure 11.1) , object β on processor A wants to send a message to a object α. Object α has its home on processor B but currently resides on processor C. If processor A has no idea where object α resides, it sends the message to the home processor of object α,

ie processor B.

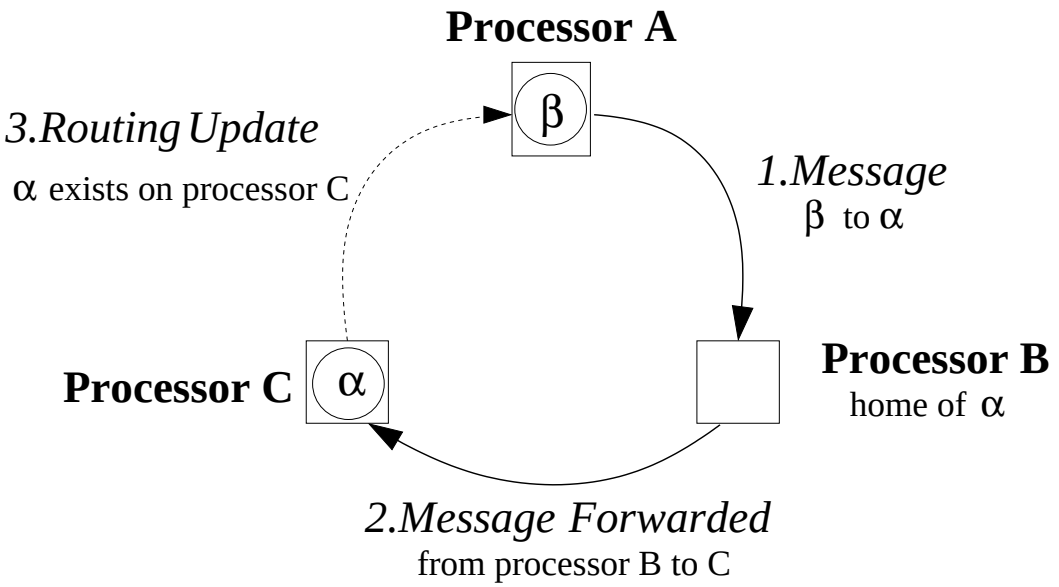


Figure 11.1: Message being sent from object β to object α. Object α exists on processor C, whereas its home is on processor B. Processor A on which object β exists does not know where α exists.

Processor B, being the home, knows that object α can be reached on processor C. This means that either processor C contains object α or knows where object α resides. Processor B forwards the message to processor C. Processor C contains object α and can hand over the message. Since forwarding is inefficient, we do not want subsequent messages for object α from objects on processor A to be forwarded through processor B. Therefore, processor C sends a routing update to A, advising it to send future messages for object α directly to C.

The situation is complicated slightly by migration. If a processor receives a message for an object that has migrated away from it, the message is forwarded to the object's last known location. Figure 11.2 illustrates the case when an object β on processor A tries to send a message to object α on processor C. At the same time object α migrates from processor C to another processor D. A migration update is sent from processor C to processor B, the home processor of

object α, telling B that α is migrating to processor D. However, in the situation shown in Figure 11.2 processor B receives the message from object β before it receives the migration update from processor C. Therefore, processor B forwards the message from object β to processor C.

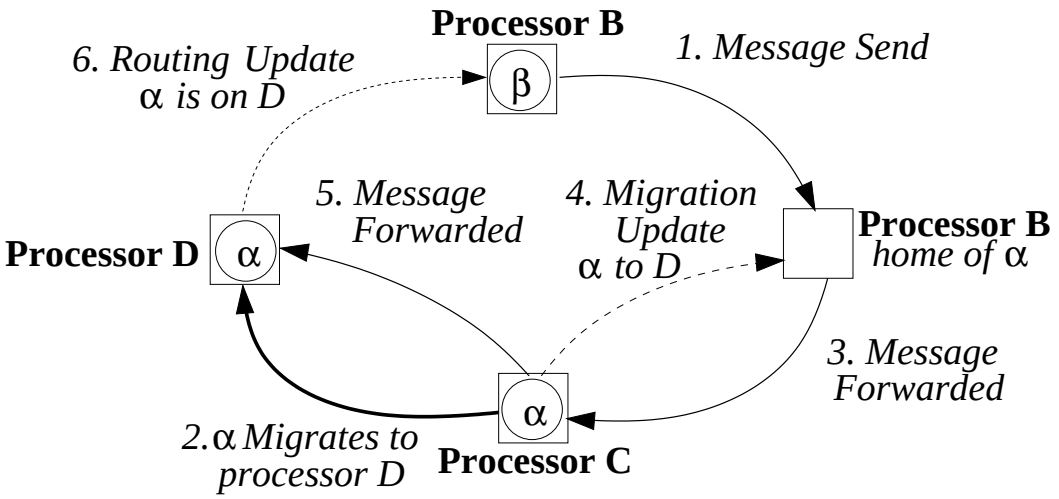


Figure 11.2: Message from object Y to X while X migrates from processor C to D.

Although, processor C does not contain object α, it knows that object α has been sent to processor D. So, it forwards the message from object β to processor C. By the time the forwarded message gets to processor D, object α has been recreated on it. Therefore object α can now processor the message from object β. Processor D notes that this message that started out from processor A, was forwarded. Therefore, processor D sends a routing upgrade to processor A, telling it that object α exists on processor D. Any subsequent message for object α from an object on processor A, will be sent to processor D directly. Processor B also receives the migration update from processor C and forwards any future messages for object α to processor D. The protocol is discussed in much greater detail in [39] describes the protocol in much greater detail.

When a processor detects that failure is imminent, it can evacuate the

CHARM++ objects located on it, to other processors. This makes sure that even if the warned processor were to crash, there are no CHARM++ objects that disappear along with it. However, this is not sufficient to ensure that an application can continue with its execution if the warned processor crashes.

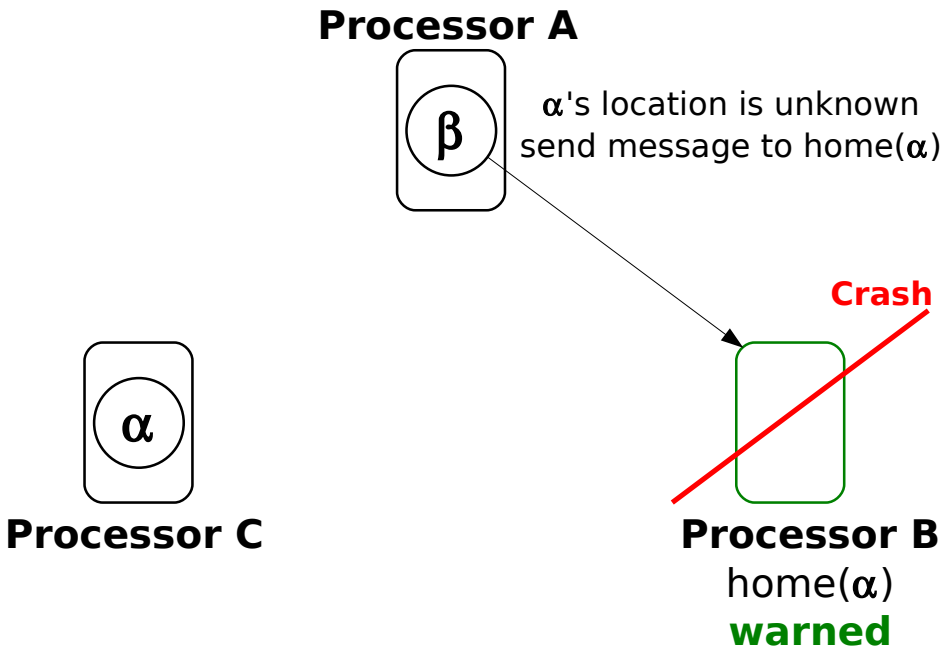


Figure 11.3: Message from object β to object α can become missing once processor B, the home of α is evacuated and then crashes.

Figure 11.3 illustrates a situation in which the crash of a warned processor can potentially cause the application to hang, even if there are no objects on the crashed processor. Processor B receives a warning and evacuates any objects that were located on it. Processor B is also the home processor of object α, which currently exists on processor C. Object β on processor A sends a message to object α. However, processor A does not know the location of object α. This is possible if no object on processor A has ever communicated with object α. In this situation according to the message delivery protocol discussed earlier, processor A should send the message to the home of object α namely processor

B. However, if processor B has already crashed processor A has no other way of sending the message to its destination, object α on processor C. In this situation, the message sent by object β to object α never gets delivered. This can cause errors in an application or cause it to hang.

One evident solution is for processor A to ask all the remaining processors for the location of object α. Processor C would tell processor A that it contains object α and processor A could send it the message. However, the next time object α migrates, this process would have to be repeated again. Moreover, this would have to happen for every object that had its home on processor B. So, this solution will not scale with the number of processors and would also cause a flood of messages every time an object that had its home on processor B migrates.

We solve the problem by assigning new homes to objects that had their homes on a warned processor. We assign new homes by changing the index-to-home mapping such that all objects with homes on a warned processor E now map to some other processor F. All processors in the application need to be informed of this changed mapping, so that they stop considering the warned processor E as the home of some objects. Thus, once the mapping has been changed on all processors and all the objects on processor E migrated out, message delivery can continue safely even if processor E crashes. Moreover, once objects have been mapped to their new homes, the message delivery protocol can continue as before. The new home of an object always knows how that object can be reached.

Figure 11.4 shows the messages sent after a processor E receives a warning. Once processor E receives a warning that a failure is imminent, it changes the index-to-home mapping so that all objects that previously had their home on E, now map to F. Then it sends a high priority *evacuation* message to all other

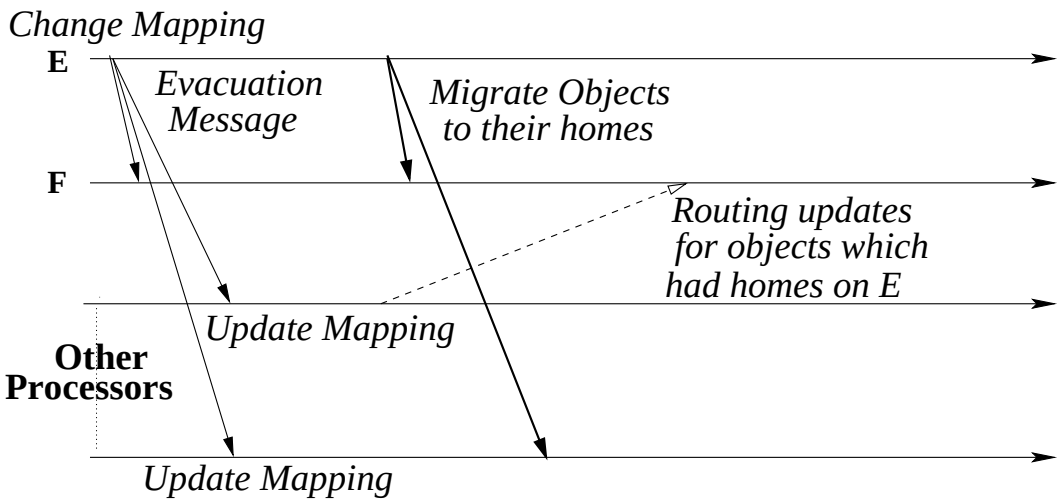


Figure 11.4: Messages exchanged when processor E is being evacuated.

valid processors (processors that have not sent *evacuation* messages to this one in the past). It also sends all objects on E to their home processors, including objects that previously had their homes on E.

When a processor other than E receives an *evacuation* message from processor E, it marks E as invalid. It changes the mapping so that all objects previously mapped to E now map to F. The mapping should be changed in such a way that all processors independently agree on the same replacement for E. For any object whose routing record points to E, change the routing records to point to that object's home processor. If this processor contains any object that previously had its home on E, inform its new home processor F about the object's current position.

The index-to-home mapping is a function that maps an object index and the set of valid processors to a valid processor. If the set of valid processors is given by the bitmap *isValidProcessor*, the initial number of processors is *numberProcessors* and *sizeOfNode* is the number of processors in a node, then an index-to-home mapping is given in Figure 11.5.

For efficiency, we derive the mapping once and store it in a hashtable for

```
start ← possible ← (index mod numberProcessors)
while !isValidProcessor[possible] do
  possible ← (possible + sizeOfNode) mod numberProcessors
  if inSameNode(start, possible) then
     abort("No valid node left")
  end
end
return possible;
```

Figure 11.5: The function to calculate the index-to-home mapping

subsequent accesses. When an *evacuation* message is received, we repopulate the hashtable.

We now discuss the protocol's behavior in different cases and whether E needs to process a message after being warned. Of course, any messages to E sent before a processor receives the *evacuation* message from E will have to be processed or forwarded by E. There is no way around it, although the high priority of the *evacuation* message tries to reduce the number of such messages.

We first analyze the effect of this algorithm on objects that had their homes on E. This protocol assigns a new home F for all such objects (let α be one of them). If object α were on processor E, it is migrated to processor F; if it existed on other processors, F is informed of its current position. If processor F receives a message for object α after having received the *evacuation* message from processor E, but before α has migrated into it or it has been informed of object α's new position, the message is buffered. When either object α or its position update is received, the buffered messages are either processed locally or forwarded to the location of α. Any messages for object α received after this follow the basic protocol. Thus, no messages are sent to processor E in this case.

If a processor sends to processor F a message for object α before F has received the *evacuation* message from processor E, F has no option but to send

it to either E or some other processor which has previously told F that α exists on it. In this case, it is possible that E would have to process a message after receiving a fault warning.

Any object (say γ) existing on E but having its home on some other processor (say G) is sent to its home processor G. The *evacuate* message changes the routing tables of all processors such that they will send all messages for object γ to processor G, instead of sending to processor E. If any message for object γ gets to processor G before γ itself, but after the *evacuate* message, it is buffered. Again the only case in which processor E might receive a message is if processor G has not received the *evacuate* message when it receives a message for object γ. All objects on processor E are sent to their home processors and not other processors because, in this case, E does not need to send a migration update to the home processors of the objects. The objects themselves fullfil that purpose. Any old routing entries pointing to processor E for an object that actually does not exist on E are updated to point to that object's home processor. Thus, according to this protocol, processor E might have to forward some messages sent or forwarded by other processors before they had received the *evacuation* message. Once all processors have received the *evacuation* message, no messages destined for Charm++ objects will be sent to processor E.

This protocol is robust enough to deal with multiple simultaneous fault warnings. The distributed nature of the algorithm, without any centralized arbitrator or even a collective operation, makes it robust. The only way two warned processors can interfere with each other's evacuation is if one of them (say H) is the home for an object existing on the other (say J). This might cause J to evacuate some objects to H. Even in this case once J receives the *evacuation* message from H, it changes its index-to-home mapping and does not evacuate objects to H. Only objects that J evacuates before receiving an

evacuation message from H are received by H. Though H can of course deal with these by forwarding them to their new home, this increases the evacuation time. This case might occur if H receives J's *evacuation* message before it receives its own warning and so does not send an evacuation message to J. We reduce the chances of this by forcing a processor to send an *evacuation* message to not only all valid processors but also processors that started their evacuation recently.

11.1.2 Support for Collective Operations in the Presence of Fault Warnings

Collective operations are important primitives for parallel programs. It is essential that they continue to operate correctly even after a crash. Asynchronous reductions are implemented in Charm++ by reducing the values from all objects residing on a processor and then reducing these partial results across all processors [39]. The processors are arranged in a k-ary reduction tree. Each processor reduces the values from its local objects and the values from the processors that are its children, and passes the result along to its parent. Reductions occur in the same sequence on all objects and are identified by a sequence number. If a processor were to crash, the tree could become disconnected. Therefore, we try to rearrange the tree around the tree node corresponding to the warned processor. If such a node is a leaf, then rearranging the tree involves just deleting it from its parent's list of children. In the case of an internal tree node, the transformation is shown in Figure 11.6. Though this rearrangement increases the number of children for some nodes in the tree, the number of nodes whose parent or children change is limited to the node associated to the warned processor, its parent and its children.

Since rearranging a reduction tree while reductions are in progress is very

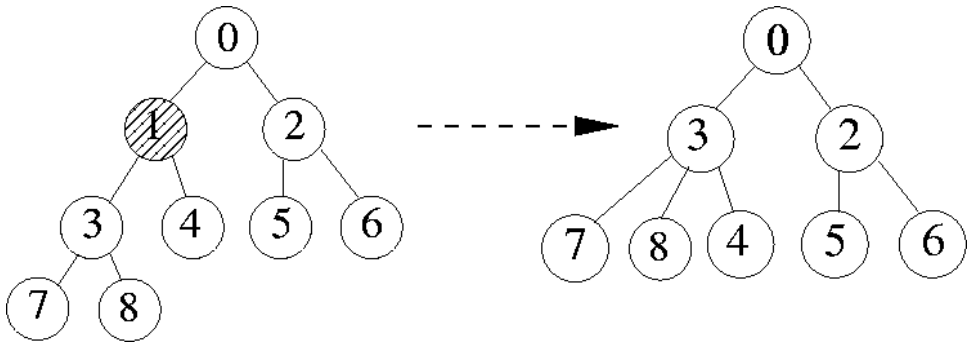


Figure 11.6: Rearranging of the reduction tree, when processor 1 receives a fault warning.

complicated, we adopt a simpler solution. The node representing the warned processor polls its parent, children and itself for the highest reduction that any of them has started. Because the rearranging affects only these nodes, each of them shifts to using the new tree when it has finished the highest reduction started on the old tree by one of these nodes. If there are warnings on a node and on one of its children at the same time, we let the parent modify the tree first and then let the child change the modified tree. Other changes to the tree can go on simultaneously and do not need to be ordered amongst each other.

The exact sequence of messages is the following:

1. Warned processor sends the tree modifications to parent and children.

2. Parent and children store the changes but do not apply them to the current tree. They reply with the highest reduction number that they have seen. They also buffer any further reduction messages.

3. The warned processor finds the maximum reduction number and informs the parent and children.

4. The parent and children unblock and continue until they reach the maximum reduction number; at that point, they change to the new tree.

The Charm++ runtime provides support for asynchronous broadcasts to its objects [39]. It simplifies the semantics of using broadcasts by guaranteeing that all objects receive broadcasts in the same sequence. All broadcasts are forwarded to an appointed *serializer*. This processor allots a number to a broadcast and sends it down the broadcast tree to all other processors. Each processor delivers the broadcast messages to the resident objects in order of the broadcast number. Contrary to intuition, this does not create a *hotspot* since the number of messages received and sent by each processor during a broadcast is unchanged.

We can change the broadcast tree in a way similar to the reduction tree. However, if the serializer receives a warning we piggyback the current broadcast number along with the *evacuation* message. Each processor changes the serializer according to a predetermined function depending on the set of valid processors. The processor that becomes the new serializer stores the piggybacked broadcast count. Any broadcast messages received by the old serializer are forwarded to the new one.

It is evident from the protocol that evacuating a processor might lead to severe load imbalance. Therefore, it is necessary that the runtime system be able to balance the load after a migration caused by fault warning. Minor changes to the already existing Charm++ load balancing framework allow us to map the objects to the remaining subset of valid processors. As we show in Section 11.2, this capability has a major effect on performance of an application.

11.1.3 Processor Evacuation in AMPI

We modified the implementation of AMPI to allow the runtime system to migrate AMPI threads even when messages are in flight, i.e. when there are outstanding MPI requests or receives. This is done by treating outstanding

requests and receives as part of the state of an AMPI thread. When a thread migrates from processor A to B, the queue of requests is also packed on A and sent to processor B. At the destination processor B, the queue is unpacked and the AMPI thread restarts waiting on the queued requests. However, just packing the requests along with the thread is not sufficient. Almost all outstanding requests and receives are associated with a user-allocated buffer where the received data should be placed. Packing and moving the buffer from A to B might cause the buffer to have a different address on B's memory. Hence the outstanding request that was copied over to the destination would point to a wrong memory address on B. One could try to update the buffer address in the request. This would require checking the request's buffer address against the old address of every user allocated buffer on the migrated thread. This check could be very costly. Another possible solution would require the user to inform the runtime system about the association between his buffers and requests during unpacking. Since this would require extra user code, we rule out the second solution as well.

We solve this problem by using the concept of *isomalloc* proposed in PM^2 [7]. AMPI already uses this to implement thread migration. We divide the virtual address space equally among all the processors. Each processor allocates memory for the user only in the portion of the virtual address space alloted to it. This means that no two buffers allocated by the user code on different processors will overlap. This allows all user buffers in a thread to be recreated at the same address on B as on A. Thus, the buffer addresses in the requests of the migrated thread point to a valid address on B as well. This method has the disadvantage of restricting the amount of virtual address space available to the user on each processor. However, this is a drawback only for 32-bit machines. In the case of 64-bit machines, even dividing up the virtual address space leaves

more than sufficient virtual address space for each processor.

11.2 Experimental Results

We conducted a series of experiments to assess the effectiveness of our task migration technique under imminent faults. We measured both the response time after a fault is predicted and the overall impact of the migrations on application performance. In our tests, we used a 5-point stencil code, written in C and MPI, and the *Sweep3d* code, which is written in Fortran and MPI. The 5-point stencil code allows a better control of memory usage and computation granularity than a more complex application. Sweep3d is the kernel of a real ASCI application; it solves a 3D Cartesian geometry neutron transport problem using a two-dimensional processor configuration.

We executed our tests on NCSA's Tungsten system, a cluster of 3.2 GHz dual-Xeon nodes, with 3 GBytes of RAM per node, and two kinds of interconnects, Myrinet and Gigabit-Ethernet. Each node runs Linux kernel 2.4.20-31.9. We compiled the stencil program with GNU GCC version 3.2.2, and the Sweep3d program with Intel's Fortran compiler version 8.0.066. For both programs, we used AMPI and Charm++ over the Myrinet and Gigabit interconnects. We simulated a fault warning by sending the USR1 signal to an application process on a computation node.

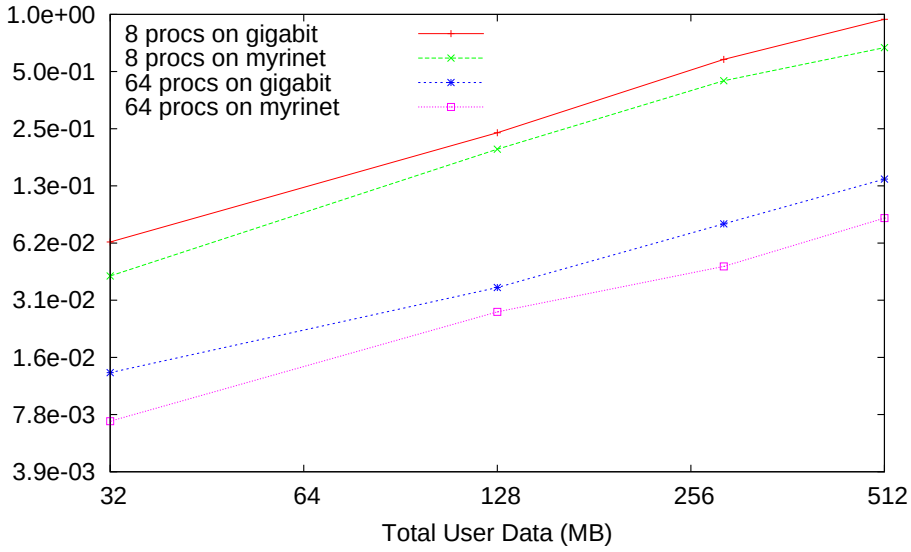
11.2.1 Response Time Assessment

We wanted to evaluate how fast our protocol is able to morph the runtime system such that if the warned processor crashes, the runtime system remains unaffected. We call this the *processor evacuation* time. However, it is not evident how this should be exactly measured. One way to measure this value is

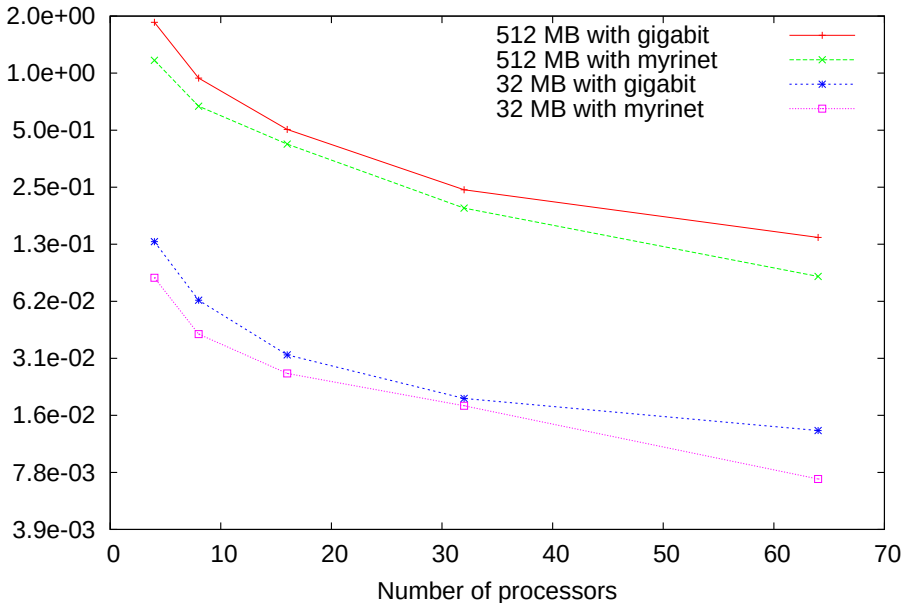
to measure the time taken to handle all messages that need to be processed by the warned processor before the runtime system can survive a fault. This includes messages for reconstructing the reduction and broadcast trees and messages sent to the warned processors before the senders received the *evacuation* message. However, this does not include the time taken for the objects on the warned processor to be actually received on the destination processors. Since clocks on different processors might not be exactly synchronized, we do not know when the objects are actually received on the destination processors. We decided to measure this time by having all the other processors send an acknowledgment back to the warned one after receiving all their objects. Thus, for a certain execution, we estimate the processor evacuation time as the maximum of the time taken to receive acknowledgments that all evacuated objects have been received at the destination processor and the last message processed by the warned processor. It should be noted that these acknowledgment messages are not necessary for the protocol; they are needed solely for evaluation. The measured value is, of course, a pessimistic estimate of the actual processor evacuation time, because it includes the overhead of those extra messages.

The processor evacuation time for the 5-point stencil program on 8 and 64 processors, for different problem sizes and for both interconnects, is shown in Figure 11.7(a). The evacuation time increases linearly with the total problem size until at least 512 MB. This shows that the evacuation time is dominated by the time to transmit the data out from the warned processor. For the same reason, the processor evacuation time for Myrinet is significantly smaller than that for Gigabit Ethernet. However, our method of measurement is biased against faster interconnects, since the measurement overheads form a more significant part of the evacuation time than in the case of slower interconnects. Hence the actual performance gain of Myrinet, in comparison to Gigabit-Ethernet, is even

better.



(a) Scaling Evacuation time with data size



(b) Scaling Evacuation time with processors

Figure 11.7: Processor evacuation time for MPI 5-point stencil calculation

Figure 11.7(b) presents the processor evacuation time for two problem sizes, 32 MB and 512 MB, of the 5-point stencil calculation on different numbers of processors. For both interconnects, the evacuation time decreases more or less

linearly with the data volume per processor. Myrinet has a significantly faster response time than Gigabit. Table 11.1 shows similar data corresponding to the evacuation time for Sweep3d, for a problem size of 150×150×150. These experiments reveal that the response to a fault warning is constrained only by the amount of data on the warned processor and the speed of the interconnect. In all cases, the evacuation time is under 2 seconds, which is much less than the time interval demanded by fault prediction as reported by other studies [45]. The observed results show that our protocol scales to at least 256 processors. In fact, the only part in our protocol that is dependent on the number of processors is the initial *evacuate* message sent out to all processors. The other parts of the protocol scale linearly with either the size of objects or the number of objects on each processor.

Number of Processors	Evacuation Time (s)
4	1.125
8	0.471
16	0.253
32	0.141
64	0.098
128	0.035
256	0.025

Table 11.1: Evacuation time for a 150^3 Sweep3d problem on different numbers of processors

11.2.2 Overall Application Performance

We evaluated the overall performance of the 5-point stencil and Sweep3d under our task migration scheme in our second set of experiments. We were particularly interested in observing how the presence of warnings and subsequent task migrations affect application behavior.

We ran the 5-point stencil application twice on 8 processors with a dateset

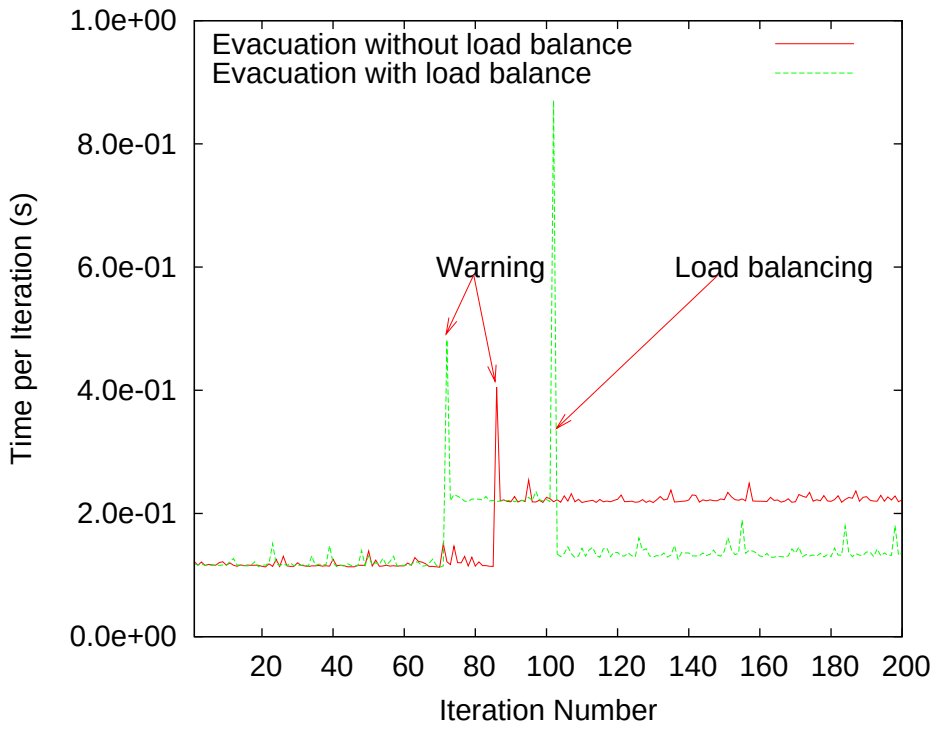


Figure 11.8: 5-point stencil with 288MB of data on 8 processors

size of 288 MB: without and with load balancing. We generated one warning during both runs. Figure 11.8 plots the time taken during each iteration. In the first execution, the evacuation prompted by the warning at iteration 85 forces the tasks in the warned processor to be sent to other processors. The destination processors become more loaded than the others. This load imbalance increases the iteration time significantly as can be seen from the red line in Figure 11.8.

We introduce a warning at iteration 70 in the second run. The green line in Figure 11.8 shows that for this run the performance immediately after the warning is the same as the first run. However, the load balancing step at iteration 100 improves the performance of the application significantly. It balances the load among the remaining processors by re-distributing the CHARM++ objects among them. After load balancing, the performance loss due to the failure warning is proportional to the computational capability that was lost (one out

of the original 8 processors was lost).

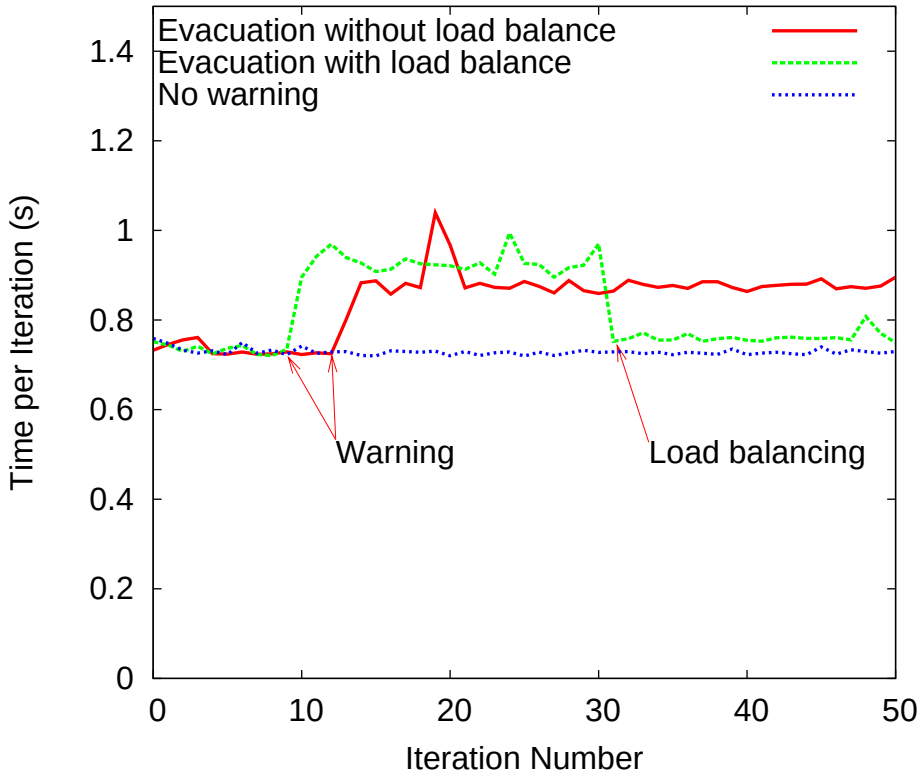
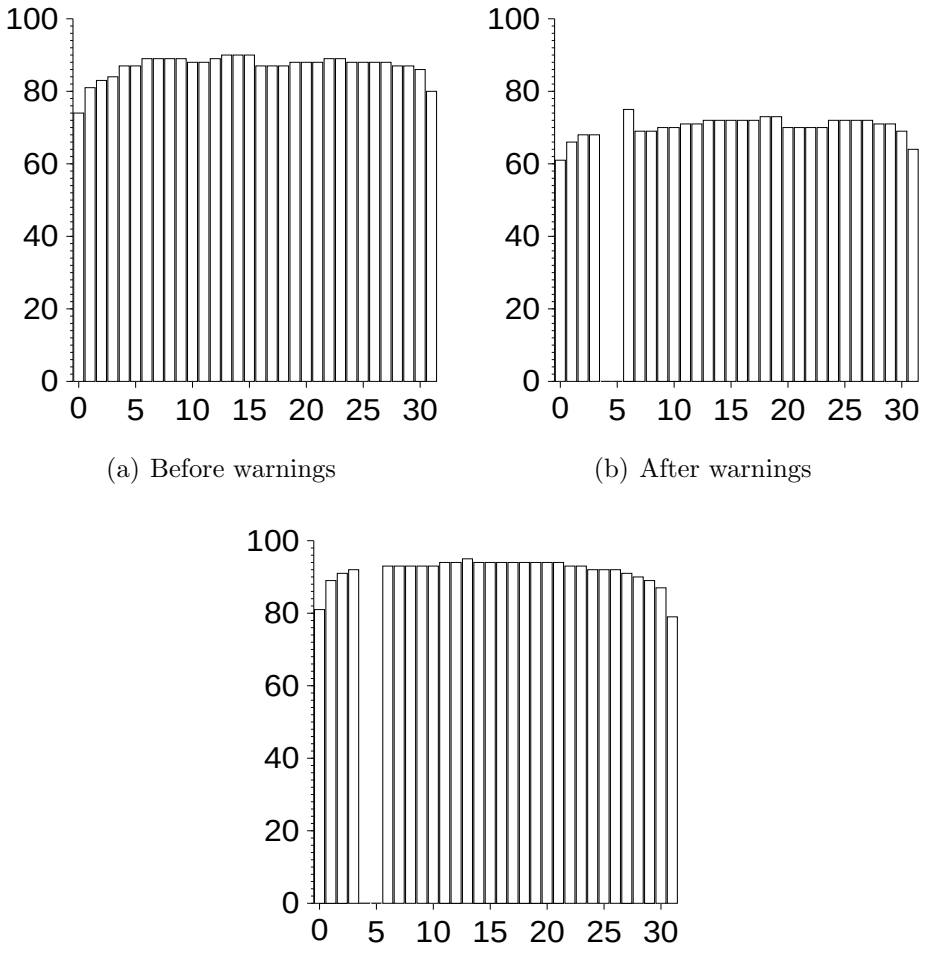


Figure 11.9: 150^3 Sweep3d problem on 32 processors

We did a similar test on the Sweep3d application. We ran Sweep3d with the 150^3 size problem thrice on 32 processors: 1) without either warning or load balancing 2) with warning but without load balancing and 3) with warning followed by a load balancing. Figure 11.9 shows the time per iteration for all three runs.

The performance of the three runs are very similar before any warnings are received. After a warning, the performance detoriates for both the runs with warnings. The iteration time for the red and green lines increase by more than 13%. This performance penalty is far more than the loss in computation power of about 3%. As before computation and communication load imbalance among the remaining processors causes this performance degradation. Once

a load balancing step is performed the performance improves markedly. The iteration time for the green line goes down sharply after load balancing. The iteration time after load balancing is only about 4% more than the iteration time before the warning. Thus the loss in performance is very similar to the loss in computation power once AMPI has performed load balancing.



(a) Before warnings

(b) After warnings

(c) Post load balancing

Figure 11.10: Utilization per processor for the 150^3 Sweep3d on 32 processors.

The Projections analysis tool processes and displays trace data collected during application execution. We use it to assess how parallel processor utilization changes across a Sweep3d execution of a 150^3 problem on 32 processors. We trigger warnings on Node 3 which contains two processors: 4 and 5 (num-

bering starts from 0). This tests the case of multiple simultaneous warnings by evacuating processors 4 and 5 at the same time.

Figure 11.10 depicts processor utilization in all 32 processors during three distinct phases. Before the warnings occur, processors have nearly uniform load and similar utilizations (Figure 11.10(a)). Processors 4 and 5 evacuate all the objects on them after receiving a warning. Figure 11.10(b) shows the utilization percentage of the different processors after the evacuation. Processors 4 and 5 have zero utilization during this period. As expected, the evacuation creates a load imbalance among the remaining processors, with some taking longer than others to finish iterations. The redistribution of objects can also increase communication load by placing objects that communicate frequently on different processors. These effects show up as low utilization for the remaining 30 processors in Figure 11.10(b).

Finally, after load balancing, the remaining processors divide the load more fairly among themselves and objects that communicate frequently are placed on the same processor, resulting in a much higher utilization (Figure 11.10(c)). The utilization for Processors 4 and 5 is still zero, showing that no objects get mapped to these processors during the load balancing. These experiments verify that our protocol matches the goals laid out at the beginning of this chapter.

We have presented a new technique for proactive fault tolerance in MPI applications, based on the task migration and load balancing capabilities of Charm++ and AMPI. When a fault is imminent, our runtime system proactively attempts to migrate execution off that processor before a crash actually happens. This processor evacuation is implemented transparently to the application programmer. Our experimental results with existing MPI applications show that the processor evacuation time is close to the limits allowed by the amount of data in a processor and the kind of interconnect. The migration

performance scales well with the dataset size. Hence, the fault response time is minimized, as required in our specifications desribed earlier in this chapter. Our experiments also demonstrated that MPI applications can continue execution despite the presence of successive failures in the underlying system. Load balancing is an important step to improve parallel efficiency after an evacuation. By using processor virtualization combined with load balancing, our runtime system was able to divide the load among the remaining fault-free processors, and application execution proceeded with optimized system utilization.

We are currently working to enhance and further extend our technique. We plan to bolster our protocol so that in the case of false positives it can expand the execution back to wrongly evacuated processors. We will also extend our protocol to allow recreating the reduction tree from scratch. We plan to investigate the associated costs and benefits and the correct moment to recreate the reduction tree.

Chapter 12

Conclusions and Future Work

We combined the ideas of message logging and object based virtualization to develop a fault tolerance protocol that provides fast recovery. Message logging let us recover from a crash without rolling back all processors to their previous checkpoints. Object based virtualization enabled us to distribute the objects on the recovering processor among the other processors in the system, thus parallelizing the recovery process and speeding it up. We modified the standard sender based message logging protocol to allow it to work it with object based virtualization. We implemented our protocol in the Charm++ environment to exploit its virtualization based run time system. We performed experiments to prove that our fast restart protocol speeds up recovery compared to basic message logging or checkpointing based schemes.

We investigated the performance penalty imposed by our fault tolerance protocol on different classes of applications. Object based virtualization turned out to be very useful again. The adaptive overlap of communication and computation provided by virtualization helped in hiding the overheads imposed by our protocol and reducing the performance penalty. We found that our protocol had comparatively higher overhead for fine grained applications. We developed optimizations that combined protocol messages to amortize the overhead and significantly reduce the cost of our protocol. We looked at the memory overhead imposed by our protocol and found ways of reducing the overhead without paying too high a cost.

Charm++ supports dynamic load balancing via object migration. However, we showed that load balancing interferes with the fault tolerance protocol. We extended the load balancing framework of the Charm++ runtime system to integrate it with our fault tolerance protocol. This lets an application utilize the load balancing framework when running a load imbalanced application with our message logging protocol. Moreover, the load balance framework is very useful in re-balancing the load after a fast recovery since, the fast restart protocol changes the object to processor mapping during recovery. Our experiments showed that using the fast restart protocol along with load balancing allows an application to complete faster than while using a simple message logging or checkpoint based protocol.

We presented proofs for the correctness of the different parts of our protocol. Our analysis showed that the protocol decreases the chances of catastrophic failure without assuming the existence of an idealized stable storage. A simple model was developed to identify the situations in which our protocol is more effective than traditional fault tolerance protocols.

The thesis included work on a proactive fault tolerance scheme that tries to evacuate processors in anticipation of a fault. We modified the Charm++ runtime system such that if a warned processor were to crash later, the application could continue execution on the remaining processors. We ensured that the run-time system continued to function efficiently by modifying the reduction and broadcast trees used by collective operations.

12.1 Limitations

Our protocol still has a number of drawbacks: some of them inherent to our design and others that can be alleviated by future work. The sender side pes-

simistic logging protocol has significant overhead for fine grained applications. Although our optimizations and object based virtualization reduce the performance penalty to a great extent, it might still be too high for some fine grained applications in certain situations. A user can use the analysis in Section 6.1 to decide whether our protocol is suitable for his application on a particular machine. In some cases, a traditional checkpoint protocol might be more suitable for a fine grained application than our fast recovery protocol

The memory overhead of our message logging protocol is another potential barrier to its adoption. The stored message logs can greatly increase the memory consumed by an application. Moreover, storing the checkpoint in the memory of another processor further increases the memory consumption as does load balancing. However, compressing the checkpoint does help reduce the memory overhead. Local disks might also be used for storing checkpoints or message logs if they grow beyond a certain size. Still, an application that barely fits into memory without any fault tolerance protocol can not really use our fault tolerance protocol. A disk based checkpoint protocol seems to be the only possible solution for such a problem.

Our protocol is designed to reduce the chances of a set of catastrophic failures that force an application to abort. It can recover from all single processor failures and a large number of multiple failures. However, if a processor and its buddy fail within the same checkpoint period then recovery becomes impossible. Thus, we do not provide any guarantees that our protocol will let an application continue no matter how many and which processors crash. We aim to build a more reliable system out of unreliable components without assuming any idealized stable storage or providing any cast-in-iron guarantees.

A limitation of our protocol in its current form is that it requires a pool of extra processors from which some can be used when some of the original

processors involved in a run crash. Removing this need would require us to carry out the recovery of the objects on the crashed processors on some of the remaining processors. Moreover, we would have to change the run time system so that it can continue to function without involving the crashed processor. In addition, we would have to remap the buddies of processors such that every processor has a buddy and no processor is the buddy of two processors. This remapping would have to be done in such a way that the reliability of the overall protocol does not get reduced. We do not yet have a complete solution for these problems though some of the solutions already described can be leveraged.

12.2 Future Work

We now see how we might be able to overcome some of the limitations described above. Our optimizations to reduce the performance penalty currently require user input. We could try to develop adaptive schemes that would decide on the buffer sizes and time out periods to use while combining protocol messages. This would not only relieve the user of trying to find out the best parameters but would also be better able to react to a program with dynamically varying communication characteristics.

Another method of reducing the overhead of our fault tolerance protocol would be to replace the sender based message logging protocol with a causal logging protocol. This could reduce the performance penalty caused by the increased latency of the message logging protocol. However, it would complicate the recovery protocol and would probably require a major overhaul to the existing fault tolerance code.

The memory overhead can be reduced by modifying the CHARM++ object packing framework so that it can compress an object's state while packing it up.

We would use the streaming compression facility provided by the zlib library. This could help us greatly reduce the peak memory usage of our fault tolerance protocol by allowing us to generate compressed checkpoints as well as compress objects being migrated to other processors. We would have to study the tradeoff between time taken to compress and the benefits of having lower peak memory usage. Similarly, message logs can themselves be stored in a streaming compressed buffer. The compressed message logs can be uncompressed and sent when a processor crashes and requires messages to be re-sent. Storing checkpoints and even message logs, when they grow beyond a certain size, on local disks might be a useful direction for investigation. As disks become faster the cost of writing to local disk might not be too onerous for the message logging protocol.

We would like to combine the fast recovery protocol with some of the ideas from the proactive object evacuation protocol. This would help us eliminate the need for a pool of extra processors. However, this would also require changing the buddy relationships between processors to ensure that all of them had a buddy and no processor was the buddy of two other processors.

We developed, presented and evaluated a fault tolerance protocol that provides fast recovery. It not only meets the goal of providing fast restarts but also tries to keep the costs and overheads low. We believe that recovery time will become an important issue for the adoption of fault tolerance protocols as machines with ever more components make it certain that any application running for a significant period of time will face faults. Every application will have to perforce use some fault tolerance protocol and the execution time for an application will depend on not only its own scalability but also the speed with which its chosen fault tolerance protocol recovers from crashes. A fault tolerance protocol with fast recovery, like the one presented in this thesis, will be an

important tool as we try to speed up applications on these future machines.

References

[1] Coordinated and improved fault tolerance for high performance computing systems. http://www.mcs.anl.gov/research/cifts/.

[2] Hardware monitoring by lm_sensors. Available at http://secure.netroedge.com/-lm78/info.html.

[3] Adnan Agbaria and Roy Friedman. Starfish: Fault-tolerant dynamic MPI programs on clusters of workstations. *Cluster Computing*, 6(3):227–236, July 2003.

[4] B. Allen. Monitoring hard disks with SMART. *Linux Journal*, January 2004.

[5] Lorenzo Alvisi, E. N. Elnozahy, Sriram Rao, Syed Amir Husain, and Asanka De Mel. An analysis of communication induced checkpointing. In *Symposium on Fault-Tolerant Computing*, pages 242–249, 1999.

[6] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal, and optimal. *Software Engineering*, 24(2):149–159, 1998.

[7] Gabriel Antoniu, Luc Bouge, and Raymond Namyst. An efficient and transparent thread migration scheme in the PM^2 runtime system. In *Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586*, pages 496–510. Springer-Verlag, April 1999.

[8] Padma Apparao and Greg Averill. Firmware-based platform reliability. Intel white paper, October 2004.

[9] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[10] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, and J. N. Seizovic. Myrinet: A gigabit per second local area network. *IEEE Micro*, 15:29–36, 1995.

[11] Eric Bohm, Glenn J. Martyna, Abhinav Bhatele, Sameer Kumar, Laxmikant V. Kale, John A. Gunnels, and Mark E. Tuckerman. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems (to appear)*, 2007.

[12] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under unix. In *ACM Transactions on Computer Systems*, pages 1–24, February 1989.

[13] George Bosilca, Aurelien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fedak, Cedile Germain, Thomas Herault, Pierre Lemarinier, Oleg Lodygensky, Frederic Magniette, Vencent Neri, and Anton Selikhov. Toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of SC 2002*. IEEE, 2002.

[14] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette. MPICH-V2: A fault tolerant MPI for volatile nodes based on the pessimistic sender based message logging programming via processor virtualization. In *Proceedings of SC'03*, November 2003.

[15] Aurelien Bouteiller, Boris Collin, Thomas Herault, Pierre Lemarinier, and Franck Cappello. Impact of event logger on causal message logging protocols for fault tolerant mpi. In *IPDPS'05*, page 97, 2005.

[16] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Automated application-level checkpointing of mpi programs. In *Principles and Practice of Parallel Programming*, June 2003.

[17] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[18] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. In *ACM Transactions on Computer Systems*, pages 3(1):63–75, February 1985.

[19] Yuqun Chen, James S. Plank, and Kai Li. Clip: A checkpointing tool for message-passing parallel programs. In *Proc. of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–11, 1997.

[20] W. E. Cohen, R. K. Gaede, and W. D. Garrett. Interconnection network independent characterization of communication traffic in the nas benchmarks via processor performance monitoring hardware.

[21] E. N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, 1992.

[22] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, October 1996.

[23] G. E. Fagg and J. J. Dongarra. Building and using a fault-tolerant MPI implementation. *International Journal of High Performance Computing Applications*, 18(3):353–361, 2004.

[24] Graham E Fagg, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J Dongarra. Scalable fault tolerant mpi: Extending the recovery algorithm. In *Proceedings of Recent Advances in Parallel Virtual Machine and Messaging Passing Interface Users' Group Meeting Euro PVMMPI 2005*, pages pp 67–75. Springer Heidelberg, Lecture Notes in Computer Science, 2005.

[25] A. Faraj and X. Yuan. Communication characteristics in the nas parallel benchmarks. In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 729–734, 2002.

[26] W. Feng. The importance of being low power in high performance computing. *CTWatch Quarterly*, 1-3, August 2005. Available at http://www.ctwatch.org/quarterly/articles/2005/08/the-importance-of-being-low-power-in-high-performance-computing/.

[27] Filippo Gioachin, Amit Sharma, Sayantan Chakravorty, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Scalable cosmology simulations on parallel machines. In *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[28] W. Gropp and E. Lusk. Fault tolerance in message passing interface programs. *International Journal of High Performance Computing Applications*, 18(3):363–372, 2004.

[29] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba. Advanced configuration and power interface specification. ACPI Specification Document, Revision 3.0, September 2004. Available from http://www.acpi.info.

[30] Chao Huang. System support for checkpoint and restart of Charm++ and AMPI applications. Master's thesis, Dep. of Computer Science, University of Illinois, Urbana, IL, 2004.

[31] J. N. Glosli and K. J. Caspersen and J. A. Gunnels and D. F. Richards and R. E. Rudd and F. H. Streitz. Extending stability beyond cpu millennium: A micron-scalesimulation of kelvin-helmholtz instability. In *presented at SuperComputing 2007*, 2007.

[32] David B. Johnson and Willy Zwaenepoel. Sender-based message logging. In *The 7th annual international symposium on fault-tolerant computing*. IEEE Computer Society, 1987.

[33] L. V. Kalé and Sanjeev Krishnan. Charm++: Parallel programming with message-driven objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[34] Laxmikant V. Kalé. Performance and productivity in parallel programming via processor virtualization. In *First Intl. Workshop on Productivity and Performance in High-End Computing (HPCA 10)*, Madrid, Spain, February 2004.

[35] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, ICCS*, Melbourne, Australia, June 2003.

[36] Gregory A. Koenig and Laxmikant V. Kale. Optimizing distributed application performance using dynamic grid topology-aware load balancing. In *21st IEEE International Parallel and Distributed Processing Symposium*, March 2007.

[37] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[38] Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng, and Laxmikant Kale. Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22(3-4):215–235, December 2006.

[39] Orion Sky Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[40] Inseon Lee, Heon Young Yeom, Taesoon Park, and Hyoung-Woo Park. A lightweight message logging scheme for fault tolerant mpi. In *PPAM*, pages 397–404, 2003.

[41] Kevib London, Shirley Moore, Daniel Terpstra, and Jack Dongarra. Support for simultaneous multiple substrate performance monitoring, October 2005. Poster Session at LACSI Symposium 2005.

[42] Sandhya Mangala, Terry Wilmarth, Sayantan Chakravorty, Nilesh Choudhury, Laxmikant V. Kale, and Philippe H. Geubelle. Parallel adaptive simulations of dynamic fracture events. *Engineering with Computers (accepted for publication)*, 2007.

[43] A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for BlueGene/L systems. Technical Report RC23077, IBM Research, January (2004).

[44] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[45] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam. Critical event prediction for proactive management in large-scale computer clusters. In *Proceedings og the ACM SIGKDD, Intl. Conf. on Knowledge Discovery Data Mining*, pages 426–435, August 2003.

[46] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[47] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. *sc*, 00:38, 2004.

[48] Georg Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531, 1996.

[49] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[50] Y. Tamir and C. Equin. Error recovery in multicomputers using global checkpoints. In *13th International Conference on Parallel Processing*, pages 32–41, August 1984.

[51] Y. M. Wang. *Space reclamation for uncordinated checkpointing in message-passing systems.* PhD thesis, University of Illinois U-C, Aug 1993.

[52] F. C. Wong, R. P. Martin, R. H. Arpaci-Dusseau, and D. E. Culler. Architectural requirements and scalability of the nas parallel benchmarks. In *Proceedings of Supercomputing*, 1999.

[53] Gengbin Zheng. *Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[54] Gengbin Zheng, Lixia Shi, and Laxmikant V. Kalé. FTC-Charm++: An in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In *IEEE International Conference on Cluster Computing*, September 2004.

Curriculum Vitae

Sayantan Chakravorty

4103 Siebel Center,
Dept. of Computer Science,
201 N. Goodwin Ave.,

Phone (office): (217) 333-5827
Phone (mobile): (217) 369-8127
Fax : (217) 265-4035

RESEARCH INTERESTS

Software based fault tolerance in large parallel systems. Runtime support for massive simulations on big parallel machines. Development of applications that scale to large numbers of processors, particularly in the fields of cosmology and unstructured mesh simulations. Efficient parallel libraries for unstructured mesh simulations. Using multiple paradigms of parallel programming in one application.

EDUCATION

M.S. in Computer Science — May 2005
University of Illinois at Urbana-Champaign — **GPA:** 3.94/4.0
Advisor: Professor Laxmikant V. Kalé

B.Tech (Hons.) in Computer Science & Engineering May 2002

Indian Institute of Technology,Kharagpur **GPA:** 9.3/10.0

HONORS AND AWARDS

- Best Thesis in Graduating Undergraduate Class, Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, 2002
- Ranked 111/150,000 in the IIT Joint Entrance Examination, India 1998
- National Talent Search Examination Scholarship from National Council for Education, Research and Training, India, India 1996

RESEARCH EXPERIENCE

- **Research Assistant**, PPL group, CS Department, University of Illinois (Advisor: Prof. Kalé)
  - **A Fault Tolerance Protocol with Fast Recovery**: Recovery time is speeded up by developing a protocol that combines the ideas of message logging and object-based processor virtualization to redistribute the work of a recovering processor among other processors. The protocol is being implemented in the Charm++ runtime system and has been evaluated to show that it provides fast restarts and has low fault-free overhead for most applications. Preliminary results were presented at the FTPDS workshop at IPDPS in 2004 and subsequent results at IPDPS 2007. This is a part of my PhD dissertation.

– **Porting Charm++ to Infiniband** : The Charm++ runtime system has been ported to the Infiniband Verbs layer, specifically to the *libibverbs* layer provided by *OpenIB*. This work is being done keeping in mind the special requirements of Charm's message driven paradigm as opposed to MPI's message passing paradigm. The implementation is being currently evaluated with real life applications like NAMD.

– **Proactive Fault Tolerance in MPI Applications via Task Migration**: This solution uses the concepts of processor virtualization and dynamic task migration, provided by Charm++ and Adaptive MPI (AMPI), to implement a mechanism that migrates tasks away from processors which are expected to fail. This research was presented at HIPC 2006.

– **Parallel Mesh Partition and Ghost Generation for Unstructured Meshes**: This work parallelized the mesh partition and ghost generation steps of a parallel unstructured mesh framework called ParFUM. This allows ParFUM to be used with large meshes that can not even be loaded on a single node. This work was my MS thesis and was a part of a journal paper on ParFUM.

– **Cosmological Simulator scalable to thousands of processors**: Co-developing a scalable cosmological simulator (recently released as Changa) helped me study data caching, communication optimizations and load balance on large numbers of processors. A paper at VECPAR 2006 shows it scaling to thousands of processors. This work, done along with cosmologists at the University of Washington, taught me the importance and challenges of coordinating develop-

ment among researchers from different academic backgrounds and spread across multiple locations.

- **Research Assistant**, Center for Process Simulation and Design, CSE Department, University of Illinois, (Advisor Prof. Kalé)
  - **Parallelization of space-time Discontinuous Galerkin method**: This work developed a parallel version of a space-time discontinuous Galerkin finite element method. It involved researchers from a wide variety of disciplines including physics, computational geometry, graphics and parallel programming. I learnt the necessity of well thought-out specifications and code modularity while working in such a varied group of scientists.
  - **Adaptivity for Parallel Unstructured Mesh Application**: This work aimed to provide methods to dynamically refine and coarsen an unstructured mesh. The degree of refinement or coarsening is determined by the application. This work has been submitted to a journal for publication.

- **Internship**, Packaging Research Center, Georgia Tech, Summer 2001 Compared the performance of Chebyshev and Power Series Expansion Functions for interpolating data.

SELECTED GRADUATE COURSES

Combinatorial Algorithms, Parallel Architecture, Advanced Compilers, Distributed Systems, Object Oriented Software Engineering, Information Theory, Machine Learning

---

PUBLICATIONS

- **Refereed Journal Articles**

  1. HPC-Colony: Services and Interfaces for Very Large Systems, Sayantan Chakravorty, Celso L. Mendes, Laxmikant V. Kalé, Terry Jones, Andrew Tauferner, Todd Inglett and Jose Moreira, *ACM SIGOPS Operating Systems Review: Operating and Runtime Systems for High-end Computing Systems*, 40(2), April,2006

  2. ParFUM: A Parallel Framework for Unstructured Meshes for Scalable Dynamic Physics Applications, Orion Lawlor and Sayantan Chakravorty and Terry Wilmarth and Nilesh Choudhury and Isaac Dooley and Gengbin Zheng and Laxmikant Kale, *Engineering with Computers*, Volume 22, Numbers 3-4 / December, 2006

  3. Parallel Adaptive Simulations of Dynamic Fracture Events, Sandhya Mangala, Terry Wilmarth, Sayantan Chakravorty, Nilesh Choudhury, Laxmikant V. Kale and Philippe H. Geubelle, *Submitted for publication to Engineering with Computers*

- **Refereed Conference Papers**

  1. A Fault Tolerance Protocol with Fast Fault Recovery, Sayantan Chakravorty, Laxmikant V. Kale, *Proceedings of the 21st International Parallel and Distributed Processing Symposium, 2007, Long Beach California*, IEEE Press, 2007

  2. Proactive Fault Tolerance in MPI Applications via Task Migration , Sayantan Chakravorty, Celso L. Mendes, Laxmikant V. Kale, *Proceedings of International Conference on High Performance Computing, 2006*, LNCS volume 4297, page 485

3. Scalable Cosmology Simulations on Parallel Machines, Filippo Gioachin and Amit Sharma and Sayantan Chakravorty and Celso Mendes and Laxmikant V. Kale and Thomas R. Quinn, *Proceedings of 7th International Meeting on High Performance Computing for Computational Science, July 2006*

4. Comparison between Chebyshev and Power Series Expansion Functions for Interpolating Data, S. Chakravorty, S. Hwan-Min and M. Swaminathan, *10th Topical Meeting on Electrical Performance of Electronic Packaging, Boston, Massachusetts, Oct. 2001*, pp: 153-156,

- **Other Publications**

1. A Fault Tolerant Protocol for Massively Parallel Machines, Sayantan Chakravorty and L. V. Kale *FTPDS Workshop for IPDPS 2004*, IEEE Press, 2004

2. Implementation of Parallel Mesh Partition and Ghost Generation for the Finite Element Mesh framework, Sayantan Chakravorty, *Master's Thesis Dept. of Computer Science, University of Illinois, 2005*
