Universitat Autònoma de Barcelona

Escola d Enginyeria

Departament d Arquitectura de Computadors i Sistemes Operatius

Fault Tolerance in Multicore Clusters. Techniques to Balance Performance and Dependability

Thesis submitted by **Hugo Daniel Meyer** for the degree of Philosophiae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Dolores Isabel Rexachs del Rosario

Barcelona, July 2014

Fault Tolerance in Multicore Clusters. Techniques to Balance Performance and Dependability

Thesis submitted by Hugo Daniel Meyer for the degree of Philosophiae Doctor by the Universitat Autònoma de Barcelona, under the supervision of Dr. Dolores Isabel Rexachs del Rosario, at the Computer Architecture and Operating Systems Department.

Barcelona, July 2014

Supervisor

Dr. Dolores Isabel Rexachs del Rosario

Abstract

English

In High Performance Computing (HPC) the demand for more performance is satisfied by increasing the number of components. With the growing scale of HPC applications has came an increase in the number of interruptions as a consequence of hardware failures. The remarkable decrease of Mean Times Between Failures (MTBF) in current systems encourages the research of suitable fault tolerance solutions which makes it possible to guarantee the successful completion of parallel applications. Therefore, fault tolerance mechanisms are a valuable feature to provide high availability in HPC clusters.

When running parallel applications on HPC clusters the prior objectives usually are: almost linear speedup, efficient resources utilization, scalability and successful completion. Hence, parallel applications executions are now facing a multiobjective problem which is focused on improving Performance while giving Fault Tolerance (FT) support; this combination is defined as Performability.

A widely used strategy to provide FT support is rollback-recovery, which consists in saving the application state periodically. In the presence of failures, applications resume their executions from the most recent saved state. These FT protocols usually introduce overheads during applications executions.

Uncoordinated checkpoint allows processes to save their states independently. By combining this protocol with message logging, problems such as the domino effect and orphan messages are avoided. Message logging techniques are usually responsible for most of the overhead during failure-free executions in uncoordinated solutions. Considering this fact, in this thesis we focus on lowering the impact of message logging techniques taking into account failure-free executions and recovery times.

A contribution of this thesis is a Hybrid Message Pessimistic Logging protocol (HM_{PL}). It focuses on combining the fast recovery feature of pessimistic receiver-based message logging with the low overhead introduced by pessimistic sender-based message logging during failure-free executions. The HM_{PL} aims to reduce the overhead introduced by pessimistic receiver-based approaches by allowing applications to continue normally before a received message is properly saved. In order to guarantee that no message is lost, a pessimistic sender-based logging is used to temporarily save messages while the receiver fully saves its received messages.

In this thesis we propose strategies to increase the performability of parallel applications. One approach focuses on avoiding performance degradation when failure occurs by using automatic spare node management to replace failed nodes. Taking into account that the fault tolerance tasks are managed by an user-transparent middleware, we propose a methodology that allows us to determine suitable configurations of these tasks in order to lower overheads in failure-free executions. This methodology is based on analyzing the message log impact and configuring the parallel executions decreasing the introduced disturbance.

Another contribution of this thesis is a methodology that allows to obtain the maximum speedup under a defined efficiency threshold, considering the impact of message logging tasks for Single Program Multiple Data (SPMD) applications. Experiments have shown that characterizing the message logging effects in computation and communication, allows us to select a better number of processes or computational cores to be used in order to improve resource utilization while increasing performability.

Keywords: Parallel applications, fault tolerance, uncoordinated checkpoint, message logging, performance, availability.

Castellano

En Computación de Altas Prestaciones (HPC), con el objetivo de aumentar las prestaciones se ha ido incrementando el número de recursos de cómputo. Con el aumento del tamano de las aplicaciones ejecutadas en computadores de altas prestaciones, también ha aumentado el número de interrupciones como consecuencia de fallos de *hardware*. La disminución en el tiempo medio entre fallos (MTBF) en los sistemas de cómputo actuales, fomenta la propuesta de soluciones de tolerancia a fallos apropiadas que permitan la finalización correcta de aplicaciones paralelas afectadas por fallos de componentes físicos. Por lo tanto, las técnicas de tolerancia a fallos son cruciales para aumentar la disponibilidad en computadores de altas prestaciones.

Cuando son ejecutadas aplicaciones paralelas en computadores de altas prestaciones, los principales objetivos normalmente son: lograr una aceleración (*speedup*) cercana a la lineal, utilización eficiente de recursos y finalización correcta de las ejecuciones. Por lo tanto, los sistemas paralelos se enfrentan a un problema multiobjetivo que consiste en incrementar las prestaciones mientras se utilizan técnicas de tolerancia a fallos; dicha combinación en la cual se consideran prestaciones y confiabilidad se denomina en la literatura *Performability*.

Las estrategias de tolerancia a fallos basadas en *rollback-recovery* han sido extensamente utilizadas en computación de altas prestaciones. Estas estrategias se basan en salvar el estado de la aplicación y cuando ocurren fallos las aplicaciones pueden volver atrás y re-ejecutar desde el último estado guardado (*checkpoint*). Estas estrategias introducen una sobrecarga durante la ejecución de las aplicaciones en aspectos que afectan a las prestaciones, tales como una sobrecarga en el tiempo de cómputo, uso de recursos, etc.

Las estrategias de checkpoint no coordinadas permiten a los procesos salvar sus estados independientemente. Cuando estas estrategias se combinan con mecanismos de log de mensajes, se evitan problemas como el efecto dominó o la aparición de mensajes huérfanos. Las técnicas de log de mensajes son responsables de la mayor parte de la sobrecarga introducida durante ejecuciones libre de fallos. En esta tesis nos hemos centrado en disminuir el impacto de las técnicas de log de mensajes considerando tanto el impacto en las ejecuciones libres de fallos así como en el tiempo de recuperación.

Una de las contribuciones de esta tesis es un mecanismo de log denominado *Hybrid Message Pessimistic Logging* (HM_{PL}). Este protocolo se enfoca en combinar las características de rápida recuperación de los mecanismos de log pesimistas basados en el receptor con la baja sobrecarga que introducen los mecanismos de log basados en el emisor en ejecuciones libres de fallos. Sin perder las ventajas de un sistema de log de mensajes pesimista basado en el receptor, el HM_{PL} permite a las aplicaciones paralelas continuar con su ejecución inclusive cuando un mensaje no ha sido guardado totalmente en el log del receptor, y de esta manera se reducen las sobrecargas en comparación con los mecanismos de log pesimistas basados en el receptor. Para poder garantizar que no se pierde ningún mensaje, un mecanismo de log pesimista basado en el emisor es utilizado para guardar temporalmente mensajes, mientras los receptores se encargan de guardar totalmente el mensaje recibido.

Además proponemos otras estrategias que permiten incrementar la *performability* de aplicaciones paralelas. Una estrategia consiste en evitar la degradación que ocurre como consecuencia de un fallo mediante la utilización automática de nodos de repuesto (*Spare Nodes*) para reemplazar nodos que han fallado. Teniendo en cuenta que las tareas de tolerancia a fallos son gestionadas por un *middleware* transparente al usuario, hemos propuesto una metodología que permite determinar configuraciones apropiadas de las tareas de tolerancia a fallos para disminuir la sobrecarga en ejecuciones libres de fallos. Esta metodología se basa en analizar el impacto del mecanismo de log de mensajes utilizado y configurar las ejecuciones paralelas disminuyendo la perturbación introducida.

Además, hemos desarrollado una metodología para la ejecución de aplicaciones pertenecientes al paradigma *Single Problem Multiple Data* (SPMD). Dicha metodología permite obtener el máximo speedup teniendo en cuenta un umbral de eficiencia computacional definida y considerando el impacto de las tareas de un mecanismo de log de mensajes. Los resultados experimentales han mostrado que mediante la caracterización del impacto del log de mensajes en el cómputo y en las comunicaciones, es posible seleccionar un número de procesos o cores de cómputo que permitan hacer un uso eficiente de los recursos mientras se aumenta la *performability*.

Palabras Clave: Aplicaciones paralelas, tolerancia a fallos, checkpoint no coordinado, log de mensajes, prestaciones, disponibilidad.

Català

La creixent demanda de capacitat de còmput en la Computació d Altes Prestacions (HPC) és satisfeta incrementant la quantitat de recursos de còmput. Amb l augment del tamany de les aplicacions executades en computadors d altes prestacions, també ha augmentat la quantitat d interrupcions com a consequència dels errors *hardware*. La disminució del temps promig entre errors (MTBF) en els sistemes de còmput actuals, fomenta la recerca de solucions de tolerància a errors adients que permetin la finalització correcta d aplicacions paral·leles afectades pels errors. Per tant, les tècniques de tolerància a errors són crucials per augmentar la disponibilitat en computadors d altes prestacions.

Quan executem aplicacions paral·leles en computadors d altes prestacions, els principals objectius normalment són: aconseguir una acceleració (*speedup*) propera a la linial, l ús eficient de recursos i la finalització correcta de les execucions. Per tant, els sistemes paral·lels s enfronten a un problema multiobjectiu que consisteix en incrementar les prestacions mentre s utilitzen tècniques de tolerància a errors; aquesta combinació en la que es consideren prestacions i confiabilitat s anomena a la literatura *Performability*.

Les estratègies de tolerància a errors basades en *rollback-recovery* han estat àmpliament utilitzades en la computació d altes prestacions. Aquestes estratègies de tolerància a errors es fonamenten en salvar l estat de l aplicació i, quan succeeixen errors, les aplicacions poden tornar enrere i tornar a executar des de l últim estat guardat (*checkpoint*). Aquestes estratègies introdueixen una sobrecàrrega durant l execució de les aplicacions en aspectes que afecten a les prestacions, tals com una sobrecàrrega en el temps de còmput, en l ús de recursos, etc.

Les estratègies de checkpoint no coordinades permeten als processos salvar els seus estats independentment. Quan aquestes estratègies es combinen amb mecanismes de log de missatges, s eviten problemes com l efecte dominó o l aparició de missatges orfes. Les tècniques de log de missatges són responsables de la major part de la sobrecàrrega introduida durant execucions lliures d errors. En aquesta tesi ens hem centrat en disminuir l impacte de les tècniques de log de missatges considerant tant l impacte en les execucions lliures d errors, aixA com en el temps de recuperació.

Una de les contribucions d aquesta tesi és un mecanisme de log anomenat *Hybrid Message Pessimistic Logging* (HM_{PL}). Aquest protocol s enfoca a combinar les característiques de ràpida recuperació dels mecanismes pessimistes basats en el receptor, amb la baixa sobrecàrrega que introdueixen els mecanismes de log basats en l emissor en execucions lliures d errors. El HM_{PL} permet a les aplicacions paral·leles continuar amb la seva execució inclús quan un missatge no ha estat guardat totalment al log, i d aquesta forma es redueixen les sobrecàrregues en comparació amb els mecanismes de log pessimistes basats en el receptor. Per poder garantir que no es perd cap missatge, un mecanisme de log pessimista basat en l emissor és utilitzat per guardar temporalment missatges, mentre que els receptors s encarreguen de guardar totalment el missatge rebut.

També proposem estratègies que permeten augmentar la *performability* d aplicacions paral·leles. Una estratègia consisteix en evitar la degradació que succeeix com a conseqüència d un error mijançant l ús automàtic de nodes de recanvi (*Spare Nodes*) per a reemplaçar nodes que han fallat. Tenint en compte que les tasques de tolerància a errors són gestionades per un *middleware* transparent a l usuari, hem proposat una metodologia que permet determinar configuracions adients de les tasques de tolerància a errors per disminuir la sobrecàrrega en execucions lliures d errors. Aquesta metodologia es basa en analitzar l impacte del mecanisme de log de missatges utilitzat i configurar les execucions paral·leles disminuint la perturbació introduıda.

A més a més, hem desenvolupat una metodologia per l execució d aplicacions pertanyents al paradigma *Single Problem Multiple Data* (SPMD). Aquesta metodologia permet obtenir el màxim speedup tenint en compte un llindar d eficiència computacional definida i considerant l impacte de les tasques d un mecanisme de log de missatges. Experiments inicials han mostrat que mitjançant la caracterització de l impacte del log de missatges en el còmput i les comunicacions, és possible seleccionar un nombre de processos o cores de còmput que permetin fer un ús eficient dels recursos mentre que s augmenta la *performability*.

Paraules Clau: Aplicacions paral·leles, tolerància a errors, checkpoint no coordinat, log de missatges, prestacions, disponibilitat.

Acknowledgements

By presenting this thesis, I am finishing one of the best chapters of the history of my life. Many people have shared with me this great journey, and I would like to thank all of them.

First of all, I would like to thank God for the guidance and for all the blessings that I have received during all these years. The truth is that without God s support, this couldn t have been possible.

I would also like to thank my wife Noemí for her unconditional love and support. Thank you Noe for all the patience that you have had with me and for all the sacrifices that you have made for me. Thank you also for always encouraging me and believing in me. I would like to thank also my mother Suzi and my grandmother Mamina, because they have been the ones that taught me to never surrender and fight for my dreams. Thank you mom for all your support, love and for making countless sacrifices to give me a chance to study a professional career.

I would like to thank my supervisors Dolores Rexachs and Emilio Luque for the continuous support, patience, motivation, enthusiasm and immense collaboration. Their guidance helped me during all the time that I have invested in researching and writing this thesis. I could not have imagined having better advisors and mentors. Special thanks also to Remo Suppi and to all professors in the CAOS department.

I would also like to thank my fellow labmates: Carlos Nunez, Abel Castellanos, Aprigio Bezerra, Marcela Castro, Javier Martinez, Alvaro Wong, Javier Panadero, César Acevedo, Tharso Ferreira, Julio García, Eduardo Cabrera, Carlos Brún, Sandra Mendez, Pilar Gómez, Andrés Cencerrado, Claudio Marqués, Arindam Choudhury, Joe Carrion, Cecilia Jaramillo, Francisco Borges, César Allande, Joao Gramacho and all the others. I would like to give special thanks to Ronal Muresano for his friendship and for the many hours that he has spent working with me.

I must also acknowledge all the people in the Distributed and Parallel Systems Group of the University of Innsbruck for their hospitality during my research stay in Austria, specially to Thomas Fahringer, Juanjo Durillo, Malik Junaid, Vincenzo De Maio and Simon Ostermann. I want to thank all the people that have influenced my undergraduate studies at the Universidad Nacional de Asunción in Paraguay. Specially, I must thank Benjamín Barán, Oscar Parra and María Elena García for all their support.

Finally, I also want to thank all my friends here in Spain and in Paraguay for their friendship and for making this journey more enjoyable.

Contents

1 Introduction **1**

1.1 Fault Tolerance in Parallel Computing 1

1.2 Motivation 6

1.3 Objectives 8

1.4 Contributions 9

1.4.1 A new Pessimistic Message Logging Technique 10

1.4.2 Increasing Performability of Applications 10

1.4.3 Case Study: Methodology to increase Performability of SPMD Applications 12

1.5 Thesis Outline 12

2 Thesis Background **15**

2.1 Parallel Applications in Multicore Clusters 15

2.1.1 Multicore Architecture 16

2.1.2 Message Passing Interface 17

2.1.3 Main Parallel Metrics 19

2.2 Classic Fault Tolerance Mechanisms in HPC 22

2.2.1 Basic Concepts 23

2.2.2 Coordinated Checkpoint 26

2.2.3 Uncoordinated Checkpoint 28

2.2.4 Message Logging 29

2.3 RADIC Architecture 32

2.3.1 Design Keys 33

2.3.2 Components and Main Functions 34

2.3.3 RADIC Operation 36

2.3.4 Implementation Details 38

3 Related Work **43**

3.1 Performance Improvement 43

3.2 Message Logging 44
3.3 Fault Tolerance Solutions 45
3.3.1 Hybrid Protocols 45
3.3.2 Other Fault Tolerance Solutions 46

4 Improving Current Pessimistic Message Logging Protocols 53
4.1 Message Logging Description 54
4.1.1 Sender-Based Message Logging 55
4.1.2 Receiver-Based Message Logging 59
4.2 Hybrid Message Pessimistic Logging 61
4.2.1 Key Concepts 61
4.2.2 Design 62
4.2.3 Orphan Processes 66
4.2.4 Implementation Details 67
4.3 Experimental Validation 68
4.4 Discussion 69

5 Balancing Dependability and Performance in Parallel Applications 73
5.1 Parallel Applications in Multicore Environments 74
5.2 Message Logging Processes Mapping 76
5.2.1 Message Logging Tasks 77
5.2.2 Analyzing Parallel Applications 77
5.2.3 Methodology to Determine Suitable Configurations 80
5.2.4 Methodology Validation 83
5.3 Case Study: Increasing Performability of SPMD Applications 86
5.3.1 Characterization 92
5.3.2 Distribution Model 100
5.3.3 Mapping 102
5.3.4 Scheduling 105
5.3.5 Performability Framework 107
5.3.6 Validation Example 108
5.4 Discussion 109

6 Experimental Results 111
6.1 Experimental Environment 111
6.2 Operation of RADIC Fault Tolerant Architecture 113
6.2.1 Design and inclusion of Spare Nodes in RADIC 113

6.2.2 Experimental Validation 114
6.3 Hybrid Message Pessimistic Logging 118
6.3.1 Comparison of Logging Techniques in Failure-free Executions 119
6.3.2 Experimental Results in Faulty Executions 121
6.3.3 Limitations and Overhead Analysis 124
6.4 Determining suitable Fault Tolerance configurations 127
6.5 Increasing Performability of Parallel Applications 129
6.5.1 Applications 129
6.5.2 Prediction Analysis 131
6.5.3 Weak Scalability and Overhead Analysis 134

7 Conclusions 139
7.1 Final Conclusions 139
7.2 Future Work and Open Lines 141
7.3 List of Publications 142

Bibliography 145

List of Figures

1.1 Sender-Based Message Logging. 4

1.2 Receiver-Based Message Logging. 5

1.3 Expected Message Logging Performance. 9

1.4 Hybrid Message Pessimistic Logging. 11

2.1 Dual Core Nodes. 17

2.2 Quad Core Multicore Cluster. 18

2.3 Rollback-Recovery Protocols. 23

2.4 Parallel application running with Coordinated Checkpoint support. 27

2.5 Parallel application running with Uncoordinated Checkpoint support . . . 28

2.6 Message Types in uncoordinated checkpoint protocols. 30

2.7 RADIC main components. 34

2.8 RADIC Scenarios: a) Fault free execution. b) Failure in Node 7. c) Utilization of spare Node , transference of checkpoints, Heartbeat/watchdog restoration and assignation of a new protector to processes of Node 8. d) Restart of faulty processes in Spare Node. 35

2.9 Fault Tolerance Layers. 39

2.10 Dividing Protector Tasks. 42

4.1 Sender-Based Message Logging. 55

4.2 Receiver-Based Message Logging. 59

4.3 Hybrid Message Pessimistic Logging 63

4.4 Hybrid Message Pessimistic Logging. Protection mechanism. 64

4.5 Hybrid Message Pessimistic Logging. Recovery of the receiver. 64

4.6 Orphan Processes in the Hybrid Message Pessimistic Logging. 67

4.7 Bandwidth utilization obtained with NetPipe Tool. 70

4.8 Overheads in message transmissions obtained with NetPipe Tool. 70

5.1 Parallel Executions Scenarios in a SPMD App. a)Communication Bound. b) Computation and Communication overlapped. c) Computation Bound. . 75

5.2 Message Logging Impact Analysis in Parallel Applications. 76
5.3 Parallel application with RADIC fault tolerance processes. a)RADIC default configuration. b) Logger threads distributed among available cores c) Protectors processes with own computational resources. 78
5.4 Automatic Distribution of Fault Tolerance Tasks. 79
5.5 Homogenous Distribution of Fault Tolerance Tasks. 80
5.6 Saving Cores for Fault Tolerance Tasks. 81
5.7 Flowchart of the Methodology to Determine Suitable Fault Tolerance Tasks Configuration. 82
5.8 Characterization results and Overhead Analysis of Applications using Message Logging. 84
5.9 SPMD application and communication effects on multicore clusters. 88
5.10 SPMD application executed applying the Supertile Concept. 89
5.11 Heat Transfer Application. Efficiency and Speedup improvement. 90
5.12 SPMD Application Behavior using Pessimistic Receiver-based Message Logging 91
5.13 SPMD Communication Pattern Example. 93
5.14 Hierarchical Communication levels of a multicore Cluster. 94
5.15 Network characterization of the Parallel System. 96
5.16 Computation and Communication characterization of the Heat Transfer Application. 99
5.17 Logical Distribution of Supertiles among cores considering two nodes with 2 quad-core processors each. 102
5.18 Affinity of Application Processes and Fault Tolerance Processes 105
5.19 SPMD methodology summary. 106
5.20 Flowchart of the Performability Framework. 108

6.1 Failure Recovery in RADIC with and without Spare Nodes. 112
6.2 Failure Recovery in RADIC with and without Spare Nodes. 116
6.3 Comparison of overheads using the Hybrid Message Pessimistic Logging and the Pessimistic RBML considering the NAS Parallel Benchmarks. . . . 120
6.4 Comparison between the Pessimistic Receiver-based Message Logging and the Hybrid Message Pessimistic Logging in executions affected by 1 failure. 122
6.5 Breakdown of Recovery Times in executions affected by 1 failure. 123
6.6 Overhead Analysis of the CG benchmark. 125
6.7 Synthetic Application Operation. 126
6.8 Overhead Analysis with the Synthetic Application. 126

6.9 Comparison of overheads using the Hybrid Message Pessimistic Logging and the Pessimistic RBML considering the NAS Parallel Benchmarks assuming Uncoordinated Checkpoint. 129

6.10 Comparison of overheads using the Hybrid Message Pessimistic Logging and the Pessimistic RBML considering the NAS Parallel Benchmarks assuming Semicoordinated Checkpoint. 130

6.11 Computation and Communication characterization of the Heat Transfer Application and the Laplace Solver for a tile. 131

6.12 Performance and Prediction Analysis using the Methodology to improve Performability of SPMD Applications. 133

6.13 Weak Scalability Analysis of SPMD applications with Message Logging. . . 135

6.14 Overhead Analysis of SPMD applications with Message Logging. 136

List of Tables

2.1 Main tasks of RADIC components. 36

3.1 Fault tolerance in message passing parallel systems 47

5.1 Theoretical and Practical Data of Heat Transfer Application. 109

6.1 Process Mapping in an eight-node Cluster. 128

6.2 Theoretical and Practical Data of executed Applications 132

List of Equations

2.1 Availability Equation. 21

4.3 Protection Overheads of the Sender-Based Message Logging 58

4.4 Recovery Cost of the Sender-Based Message Logging 58

4.5 Protection Cost of the Receiver-Based Message Logging 60

4.6 Recovery Cost of the Sender-Based Message Logging 61

4.8 Protection Cost of the Hybrid Message Pessimistic Logging 65

4.9 Recovery Cost of the Hybrid Message Pessimistic Logging 66

5.1 Internal Computation Time for the SPMD Model 97

5.2 Edge Tile Computation Time without FT for the SPMD Model 97

5.3 Edge Computation for the SPMD Model 97

5.4 Edge Tile Communication Time for the SPMD Model 99

5.5 Communication-Computation Ratio for the SPMD Model 99

5.6 Ideal Number of Cores for an SPMD Application 100

5.10 Execution Time of a SPMD Application using Fault Tolerance 101

5.12 Ideal Overlapping between computation and communication 101

5.14 Ideal Supertile Size 101

Chapter 1

Introduction

"Only as a warrior can one withstand the path of knowledge. A warrior cannot complain or regret anything. His life is an endless challenge, and challenges cannot possibly be good or bad. Challenges are simply challenges"

Carlos Castaneda

1.1 Fault Tolerance in Parallel Computing

In recent years, computer science has become one of the most valuable tools for modern life. In this time, computer science has increasingly help the development of new essential services and applications that benefit different knowledge areas such as health, chemistry, engineering, mathematics, etc. As there is increasing interest in using computer systems to solve problems faster and to obtain better quality solutions, there is also an increasing and non-stop demand for computing power.

The increase in computing power demand is translated nowadays into application parallelization approaches and High Performance Computing (HPC) systems to execute these parallel applications. Currently, the HPC systems are being used to solve complex and computationally intensive applications such as DNA sequencing, fire forecasting, molecular dynamics simulations, weather forecasting, simulation of emergency situations, among others. In order to take advantage of HPC clusters, applications are mostly parallelized using a message passing communication paradigm such as MPI [35] which is a *de facto* standard used to write parallel applications.

The use of HPC clusters is increasingly common in science. However, the massive utilization of these parallel computers has made them grow in computation power by concentrating more components in a single machine and by combining several machines to work together. These HPC clusters are now composed of multicore machines because this gives greater computing capacity [69]. However, with this increase in the number of components, the miniaturization and high concentration, the number of failures that affects these systems also increases [40] [79].

There are several sources of failures in HPC systems, among them we can mention failures caused by Human Errors, Hardware, Network or Software faults. In [27] is presented an analysis of 22 HPC systems where it can be clearly seen that failure rates in these systems increase as the number of nodes and processors increases. From the data collected it has been determined that the main reason for stoppages in these systems are the hardware failures.

Therefore, fault tolerance mechanisms are a valuable feature to provide high availability in HPC clusters. The need for reliable fault tolerant HPC systems has intensified because a hardware failure may result in a possible increase in execution time, lower throughput and the cost of running parallel applications [27].

In spite of the current interest of researchers in fault tolerance techniques for parallel applications, this is not a new concern. These techniques have been a topic of interest for many years and it is even becoming more important because more performance improvement in systems means less reliability.

Failures impact negatively on applications performance because of the time wasted in restarting the application, repairing or replacing the failed components and re-executing the applications. If these times tend to zero, there would not be a big impact in parallel executions, but in order to achieve this fault tolerance strategies should be used. However, the usage of fault tolerance mechanisms imply an impact on cost (more resources, longer execution times, etc). Normally, in HPC replication is used to address transient failures, but when considering permanent failures, rollback-recovery strategies are used.

Rollback-recovery consists in saving states of an application periodically, and in the event of failure the application resumes its execution from the most recent saved state.

Currently, there are some well-know rollback-recovery protocols [29] [27] to handle hardware failures in parallel environments which avoid the loss of computation work. Many rollback-recovery protocols that protect parallel applications are based on checkpointing and message logging or a combination of the two.

One of the most used and implemented rollback-recovery protocol for parallel systems is the coordinated checkpoint. It consists in interrupting the whole parallel application at some points to save a global snapshot of the application, guaranteeing that no non-deterministic events are occurring (messages) in the meantime in order to obtain a consistent state [29].

In the recent past, coordinated approaches were preferred when providing fault tolerance support to parallel applications because of the low failure-free overhead that these strategies introduce and their ease of deployment. However, coordinated checkpoint approaches present three major disadvantages:

- When using fault tolerance support below the application level (application-transparent fault tolerance), increasing the number of processes involved may increase the time needed to coordinate all the processes. Normally, most protocols require at least two global coordinations [29].
- In the event of failure, all processes of the parallel application must rollback to a previous state, even the non-failed processes. Then, there may be a considerable computational time loss.
- Checkpoint and restart suffer from high I/O overhead at the scale envisioned for future systems, leading to poor overall efficiency barely competing with replication [32].

The current trend in HPC is to avoid fault tolerance solutions where collective operations (such as the coordination) and centralized components are used because they could compromise the scalability of applications. On the other hand, uncoordinated checkpoint protocols allow each process to save its state independently without needing coordination. Nevertheless, in order to avoid the domino effect [29] these protocols should be used in combination with a message logging approach.

Uncoordinated approaches guarantee that scalability is not compromised since each process may take actions on its own, avoiding the costly step of process coordination. Also, only faulty processes must rollback to a previous state reducing the amount of lost computation that has to be re-executed, and possibly permitting overlap between recovery and regular application progress.

In order to avoid the rollback of non-failed processes, uncoordinated approaches should be combined with event logging. Thus, in the event of failure restarted processes could use the logged messages without forcing other non-failed processes to rollback and recreate messages. Message Logging techniques allow faster recovery from failures, on the other hand there is a higher overhead in failure free executions because each message has to be saved in a stable storage. However, according to [52] message logging has a better application makespan than coordinated checkpoint when the Mean Time Between Failures (MTBF) is shorter than 9 hours. It is important to highlight that some message logging schemes (optimistic message logging) may force non-failed processes to rollback because

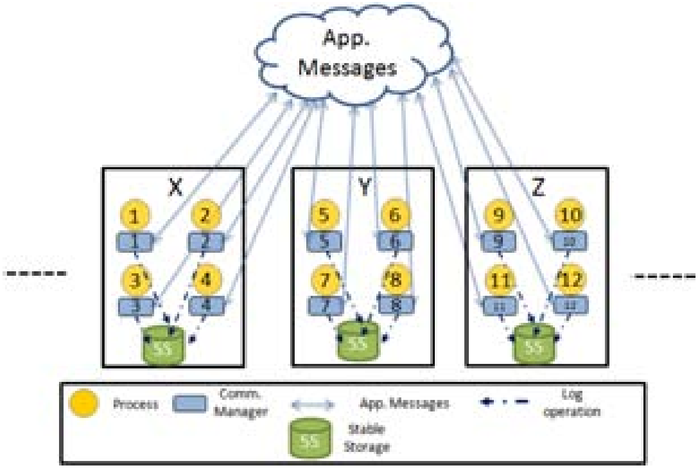


Figure 1.1: Sender-Based Message Logging.

they allow applications to continue with their executions without guaranteeing that all message events are properly saved.

There are two main message logging techniques that have been widely used and implemented in message passing systems: Sender-based Message Logging and Receiver-based Message Logging. Depending on the moment in which messages are saved to a stable storage they can be: pessimistic or optimistic.

Pessimistic approaches guarantee that messages are saved into stable storage before they can make an impact on processes not involved in the message transmission. On the other hand, optimistic approaches consider that the failure is unlikely to occur while messages are being logged, then in order to lower the overheads during failure-free executions, these approaches allow the application to continue without blocking the processes involved in the communication. When using optimistic approaches, there is a chance of creating orphan messages [29] which can lead to forced rollbacks of non-failed processes.

Figure 1.1 shows the operation of a sender-based message logging. Before sending messages, the communication managers of the senders save them into a stable storage. In the event of a failure, receivers will request its logged messages to all senders and the senders will deliver them. This approach normally introduces low overheads during failure-free executions, but when failures occur failed processes rely on the information distributed among all their senders. Then, senders should be able to re-send these messages in the appropriate order so the failed processes could reach the same before-failure state. Garbage collection in these protocols is complex, since after checkpointing each process should notify its senders to erase old messages.

Figure 1.2 depicts the operation of a receiver-based message logging. In this case when processes receive messages, they are forwarded to a stable storage. Considering

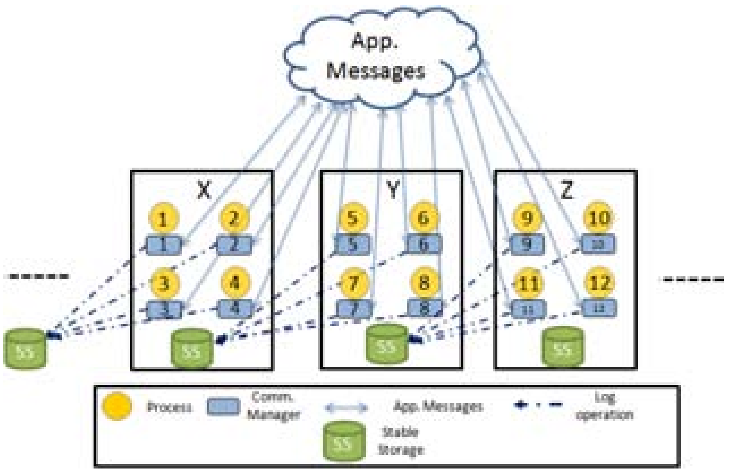


Figure 1.2: Receiver-Based Message Logging.

a distributed fault tolerant support, messages received in one node could be saved in another node. In case of failure, failed processes consume their old messages which are not distributed in several partial logs (as is the case of sender-based logging). This approach normally introduces more overhead than sender-based approaches, but on the other hand the recovery procedure is independent and faster. Garbage collection is not complex here since after a checkpoint processes could be in charge of deleting their own log.

Considering the above-mentioned facts, there are two main challenges that need to be addressed when deploying an uncoordinated checkpoint approach:

- Estimate execution time and checkpoint intervals: with coordinated protocols there are very precise models to estimate application execution time and best checkpoint intervals [23]. However, in [34] a model to estimate the most suitable checkpoint interval for parallel applications using uncoordinated protocols has been developed.

- Reduce the overhead of message logging techniques: uncoordinated approaches usually rely on the usage of a message logging approach that should save every outgoing message (sender-based logging) or incoming message (receiver-based logging). This of course, may cause a considerable negative impact on parallel execution times.

Considering that the problem of finding the best checkpoint interval for an uncoordinated checkpoint protocol has been addressed, we should focus on what happens between two uncoordinated checkpoints: *Message Logging*. Reducing the impact of message logging in parallel applications has become a very important matter since it is one of the major sources of overhead [78].

1.2 Motivation

As we have mentioned, when we increase the applications performance by using more components to execute more work or in a shorter time, reliability decreases because the probability of being affected by a failure is higher.

A recent analysis conducted in [79] has proven that the 22 different HPC systems analyzed exhibit different failure rates ranging from 20 to more than 1000 failures per year on average. Their results indicate that failure rates do not grow significantly faster than linearly with system size. Thus, failures are becoming not uncommon events, but a normal part of applications execution.

Considering the above-mentioned facts, we can assume that failure probability will continue to increase with system size. Thus, it is crucial to recognize that many long-running parallel applications may not finish correctly as a consequence of a failure.

Taking this into consideration, it is extremely important to consider that for some executions of applications, the use of fault tolerance will not be an option but a requirement in order to guarantee a successful completion. This is why it is important to look into new techniques that scale with parallel applications and that introduce less overhead while giving protection to the application allowing fast recovery. Replication of tasks may not be an option in HPC systems since only a top efficiency of 50% may be achieved, and these levels of efficiency can be improved when using rollback-recovery.

In order to provide fault tolerance support for parallel applications two main approaches are the more common. The first one is to write applications with fault tolerant support. This seems like a good option because one can select the best techniques for a specific program. However, there is a high cost software engineering and software development associated with this solution. The second common approach is to provide fault tolerance support beneath the application level, for example at the communication library level.

To include fault tolerance support at library levels comes with the advantage of application-transparency, but on the other hand the overheads of these solutions tend to be higher than the non-transparent solution.

Several studies have focused on providing MPI libraries with fault tolerance support. There are some examples in MPICH [7] [8] and in Open MPI [59] [44]. Nevertheless, it is important to investigate the development of techniques to provide dependability to parallel applications while decreasing the impact in performance and efficiency.

Given that having a centralized fault tolerant manager may compromise the scalability of the solution, in this thesis we focus on designing and include improvements into a transparent and distributed fault tolerant architecture called Redundant Array of Distributed and Independent Controllers (RADIC) [25]. However, the policies and strategies that are proposed in this thesis could also be integrated into the application layer and adapted to the needs of each application. In order to give transparent fault tolerance support and for testing purposes we have added them to RADIC.

Besides the problem of selecting the right layer to introduce fault tolerance support, it is important to consider how the fault tolerance tasks interact with the parallel application. If we consider that parallel applications are mapped into parallel environments in order to scale correctly making an efficient utilization of resources, any disturbance may render all the mapping work useless.

When single-core clusters were the only option to execute parallel applications, there were not many choices with regard to sharing resources. As there was only one computational core available, parallel applications share this resource (as well as the memory and cache levels) with the fault tolerance tasks if there were not dedicated resources. Considering that current clusters have more cores and memory levels, there is a need to develop mapping policies that allow parallel applications to coexist with the fault tolerance tasks in order to reduce the disturbance caused by these tasks. It is also important to consider that the number of cores has been multiplied by 8, 16, 32, 64 and usually the networks used in these clusters have not increase their speed to the same extent.

Several strategies have been developed to improve the performance of fault tolerance solutions, such as: rollback-recovery protocols based on message-logging that focus in lowering the overheads in failure-free executions and avoids the restart of non-failed processes [21]; diskless checkpoint approaches [39]; semi-coordinated checkpoint approaches [19] [13]. However, resource assignation and overhead management are two important matters that should be studied.

Not only has multicore technology brought more computation capability to HPC systems, it has also brought more challenges to resource assignation protocols. If fault tolerance tasks are going to coexist with application processes, then it is important to determine the best way or ways to map jobs taking into account performance, efficiency and dependability.

Considering that the MTBF is decreasing in current HPC clusters [27], uncoordinated checkpoint approaches appear to be the best option since the recovery procedure in coordinated approaches is more costly because it involves the rollback of all processes.

Since the problem of finding the best checkpoint interval for uncoordinated fault tolerance approaches was addressed in a previous work [34], this thesis focuses on describing what happens between checkpoints: Message Logging. Since the major source of overhead in fault tolerance for parallel applications is the message logging approach used [52], it is extremely important to concentrate efforts on lowering the impact of these techniques. Therefore, it is very important to design policies or optimizations to current message logging techniques in order to improve the performance metrics of fault tolerant executions.

Having turned our attention to message logging techniques, there are some questions that arise related to performance improvement while using fault tolerance:

- Is there a way to hide the negative impact on efficiency when using message logging techniques in parallel applications?
- Can we manage the resources in order to reduce the impact of fault tolerance techniques on the critical path of a fault tolerant execution?
- Are current message logging techniques good enough to satisfy current needs of availability in combination with performance improvement?

The aim of this thesis is to answer these questions by providing fault-tolerance to parallel executions in multicore clusters while reducing the negative impact on performance. This is the motivation of this thesis.

1.3 Objectives

The main aim of this thesis is to propose fault tolerance policies capable of reducing the negative impact in parallel applications while improving dependability, taking into account the advent of bigger multicore systems. In order to accomplish this goal we focus on message passing applications in HPC systems. We also consider the utilization of transparent, scalable and configurable (according to systems requirements and behavior) fault tolerance support.

Factors influencing the computer cluster s performability, such as message log latency, message log task distribution, performance degradation because of node losses are studied and solutions are presented in order to improve performability in fault-free and post-recovery situations.

In order to achieve these objectives we focus on evaluating applications behavior as well as classic fault tolerance mechanism in order to adapt them to better fit current requirements. We will focus on providing answers to the questions presented previously.

We can enumerate our objectives as follows :

1. Reduce the impact of message logging techniques in parallel executions.

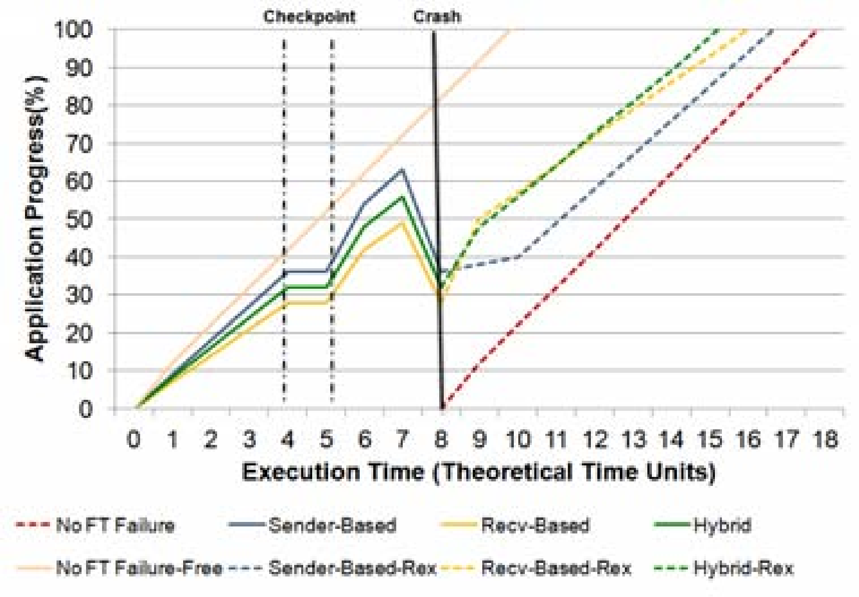


Figure 1.3: Expected Message Logging Performance.

- Propose a message logging protocol that is able to balance the trade-offs between low failure-free overheads and fast recovery. Reduce the overheads of pessimistic receiver-based message logging without increasing the complexity of the garbage collection mechanism and during recovery.

2. Integrate fault tolerance techniques efficiently into parallel systems, considering applications behavior.

   - Analyze applications behavior and characterize the impact of fault tolerance techniques, in order to map the processes into the parallel machine taking into account performance and dependability.
   - Characterize failure-free overheads in order to tune applications executions taking into account a specific kind of application. Determine the parameters that help us to find a suitable number of cores based on three main and sometimes opposing criteria: availability, the desired efficiency level of an application and a maximum speedup.

1.4 Contributions

In order to comply with the objectives established in this thesis, a new Message Logging protocol has been designed and implemented; a methodology based on characterization in order to properly select the mapping of some fault tolerance tasks has been proposed;

and an analytical method that allows to properly balance the trade-off between speedup and efficiency while using a message logging approach has been developed. In the next subsections the contributions are described in detail.

1.4.1 A new Pessimistic Message Logging Technique

In order to reduce the impact of current pessimistic receiver-based message logging approaches, we propose a new message logging protocol called Hybrid Message Pessimistic Logging (HM_{PL}). The HM_{PL} focuses on combining the fast recovery feature of pessimistic receiver-based message logging with the low protection overhead introduced by pessimistic sender-based message logging.

In Figure 1.3 we can observe the expected behaviors of message logging techniques in parallel applications. A total re-execution of the application could lead to almost double the execution time. While a sender based approach performs better during failure free-executions, receiver-based approaches penalize less in failure situations. However, the objective of the HM_{PL} is to perform better than receiver-based approaches during failure-free executions but also avoid a high penalization in case of failure.

The HM_{PL} reduces the overhead introduced by receiver-based approaches, reducing the waiting time when saving messages by using a sender-based approach that guarantees that no message is lost while allowing a parallel application to continue even before a received message is properly saved.

Figure 1.4 shows the main operation mechanisms of the HM_{PL}. As we can observe, senders save messages in a temporary buffer before sending them. These messages may be used in certain failure scenarios to allow processes to fully recover. When processes receive messages, they store these messages in a temporary buffer and continue with the normal execution without waiting for the messages to be fully saved in a stable storage at another location.

The main benefit of the HM_{PL} is that it reduces the impact in the critical path of applications by removing the blocking behavior of pessimistic approaches guaranteeing that non-failed processes will not roll back.

Contributions of this work have been published in [63].

1.4.2 Increasing Performability of Applications

Automatic Management of Spare nodes into MPI

If we consider that processes affected by node failures should be restarted in another location and there are no available nodes to execute these processes, the application can

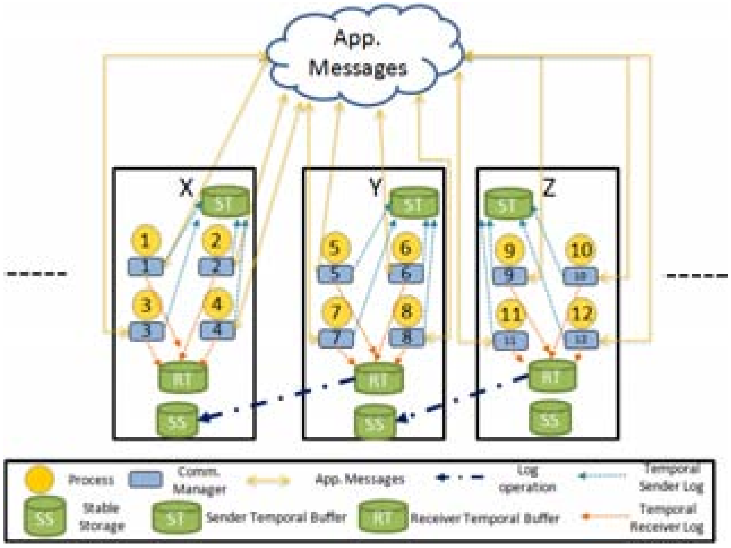


Figure 1.4: Hybrid Message Pessimistic Logging.

lose throughput as a consequence of restarting failed processes in already used nodes.

We have proposed a transparent and automatic management of spare nodes in order to avoid performance degradation of applications after the occurrence of node failures. When using this mechanism there is a slight increase in the Mean Time to Recover (MTTR), since the message log and checkpoint of the processes should be transferred to a spare node. However, this mechanism avoids the loss of computational capacity.

This work has been included as part of the Redundant Array of distributed and Independent Controllers (RADIC) in order to keep the initial process per node ratio, maintaining the original performance in case of failures. Contributions of this work have been published in [59].

Reducing Message Log Impact in Applications

In fault tolerant executions, there are fault tolerance tasks that are being executed while the applications processes are carrying out their tasks. The resource consumption of the fault tolerance tasks, in some cases, is not negligible.

Considering the advent of multicore machines, it is becoming important to propose policies to make an efficient use of the parallel environment considering the interaction between application processes and fault tolerance tasks.

To endow our message logging protocols with flexibility, we have designed our mechanisms in order to allow the distribution of log tasks among the available resources (computational cores) in a node. Our fault tolerance protocols are multithreaded, and this allows us to distribute or concentrate the overheads in computations of our message logging techniques depending on application behavior.

We focus on addressing the combination of process mapping and fault tolerance tasks mapping on multicore environments. Our main goal is to determine the configuration of a pessimistic receiver-based message logging approach which generates the least disturbance to the parallel application.

We propose characterizing the parallel application in combination with the message logging approach in order to determine the most significant aspects of the application such as computation-communication ratio and then, according to the values obtained, we suggest a configuration that can minimize the added overhead for each specific scenario. We show that in some situations it is better to save some resources for the fault tolerance tasks in order to lower the disturbance in parallel executions.

Contributions of this work have been published in [61].

1.4.3 Case Study: Methodology to increase Performability of SPMD Applications

Considering that many scientific applications are written using the SPMD (Single Program Multiple Data) paradigm, we propose a novel method for SPMD applications which allows us to obtain the maximum speedup under a defined efficiency threshold taking into account the impact of a fault tolerance strategy when executing on multicore clusters.

The main objective of this method is to determine the approximate number of computational cores and the ideal number of tiles, which permit us to obtain a suitable balance between speedup, efficiency and dependability.

This method is based on four phases: characterization, tile distribution, mapping and scheduling. The idea of this method is to characterize the effects of the added overhead of fault tolerance techniques, which seriously affect the MPI application performance. In this sense, our method manages the overheads of message logging by overlapping them with computation.

Contributions of this work have been published in [60] and [62].

1.5 Thesis Outline

Based on the objectives the remaining chapters of the thesis are structured as follows.

Chapter 2: Thesis Background.

This chapter introduces some basic concepts about fault tolerance and parallel applications in multicore environments.

Chapter 3: Related Work.

This chapter presents the related work and explains some specific concepts about message logging which is the main focus of this thesis.

Chapter 4: Improving Current Pessimistic Message Logging Protocols.

This chapter explains in detail the Hybrid Message Pessimistic Logging (HM_{PL}) approach designed for maintaining almost the same MTTR of processes compared with receiver-based message logging approaches while reducing failure-free overheads.

Chapter 5: Balancing Dependability and Performance in Parallel Applications.

In this chapter, we present the methodology to increase performability of parallel applications running on multicore clusters as well as the methodology to select the best mapping of fault tolerance tasks.

Chapter 6: Experimental Results.

This chapter describes the test scenarios and provides an explanation of the experimental results of our proposals, the Hybrid Message Pessimistic Logging (HM_{PL}), the methodology to increase performability and the methodology to select mapping of fault tolerance tasks.

Chapter 7: Conclusions.

This chapter draws the main conclusions of this thesis and presents suggestions for further work and future lines of research.

The list of references completes the document of this thesis.

Chapter 2

Thesis Background

In this chapter we present some basic concepts about parallel applications, systems and fault tolerance in High Performance Computing needed to frame the thesis. First, we introduce a general description of current parallel applications requirements in section 2.1. Then, in section 2.2 we present some concepts of fault tolerance, focusing specially in uncoordinated approaches such as message logging. Section 2.3 describes the details of the RADIC architecture which has been used as foundation of this thesis.

2.1 Parallel Applications in Multicore Clusters

The computation time of an application is directly related to the time spent in executing a set of basic instructions and to the number of concurrent operations that the system may execute.

In order to increase performance metrics, the usage of High Performance Computing (HPC) systems has increased during the last years. The main strategy consists in the usage of techniques to simultaneously process the information. In parallel computing, all processors are either tightly coupled with centralized shared memory or loosely coupled with distributed memory and the processes running collaborate to solve a computational problem (page 7 in [46]). Parallel applications divide the workload among each available process so the work can be done simultaneously in order to reduce the execution time.

The main benefit of parallel processing is the reduction of the execution time of applications. Nevertheless, the performance metrics such as scalability, speedup, efficiency, among others could be affected since in parallel environments processes compete for the shared resources (network, shared memory, storage, etc.). In multicore HPC clusters, we have several cores that use different communication channels (network, main memory, cache memory, etc.) with different bandwidths and this should be considered when dividing the workload among processes.

As parallel processes collaborate between each other to solve a computational problem, usually partial results are splitted among several processes. Therefore, it is very important that all parallel processes finish their executions correctly in order to obtain full results. However, with the increase in the number of components used, there is also an increase in the node failure probability, since the mean time between failures (MTBF) in computer clusters have become lower [13].

Consequently, we cannot avoid the fact that running an application without a fault tolerance approach has a high probability of failing and stopping when using many computational resources or when it has been executing for several hours. Now, two conflicting objectives should be taken into account: improving *Performance* while giving *Fault Tolerance* support. When protecting applications transparently with a fault tolerance technique (using the available system resources), usually an extra overhead that seriously affects the performance should be paid. In this sense, all initial tuning may be in vain when fault tolerance support is given without taking into account the influence of the overhead added on applications and even more tightly coupled applications are executed.

2.1.1 Multicore Architecture

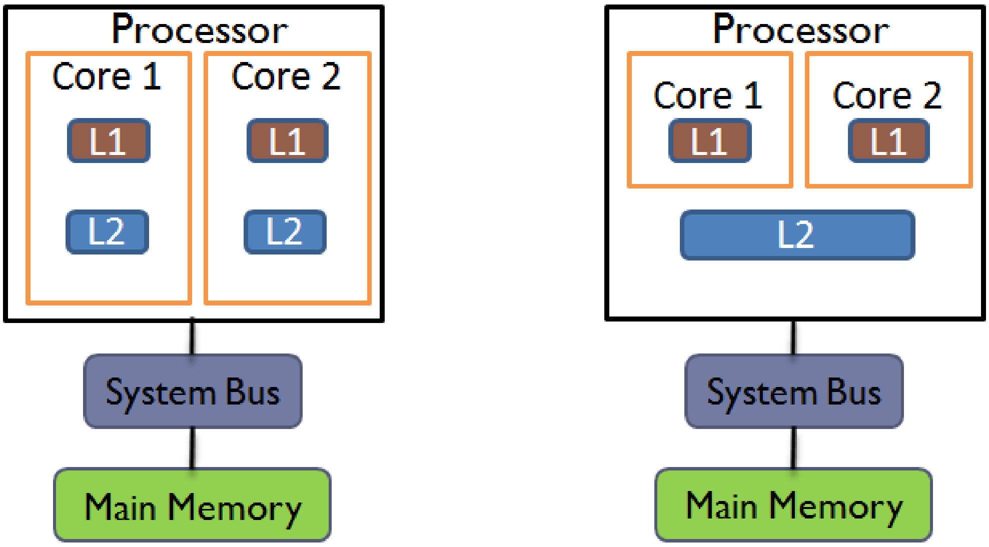
The multicore architecture is composed by two or more computational resources (cores) residing inside inside a processor [24]. The main advantage of this is that it allows to improve the performance metrics of applications by allowing the execution of multiple instructions in parallel [72].

However, when running parallel applications in multicore machines and wanting to obtain good performance metrics, some features that should be taken into account, such as: the number of cores per processor, data locality, shared memory management, hierarchical communication levels, etc.

Several alternatives are available regarding multicore node design. There are variations about the hierarchical memory definition and the way of accessing the memory. In Figure 2.1a is shown an example where the L2 cache level is individual to each core, and in Figure 2.1b is shown a design where the L2 cache is shared between two cores in a processor.

Other differences in multicore design are related to the memory access model: Uniform Memory Access (UMA) [20] and Non-Uniform Memory Access (NUMA) [47] architecture. In UMA systems, every core share uniform access to the main memory, thus the access time is the same for each of them.

On the other hand, in NUMA architectures the memory is individual for each processor that integrates the system, but processors have access to the memory spaces of other



(a) Non-shared L2 cache.

(b) Shared L2 cache.

Figure 2.1: Dual Core Nodes.

processors with a higher access time. This architecture is usually used in systems with high number of processors per node, where data contained in main memory should be controlled [15].

Current HPC clusters are composed by multicore machines, since they allow to improve the performance of applications and also reduce energy consumption.

As can be observed in Figure 2.2, a multicore cluster has an hierarchical communication system that should be taken into account when running parallel applications. There are intercore communications with shared cache (Core 1 and Core 2 of Processor 1 in Node1), without shared cache (Core 1 and Core 3 of Processor 1 in Node1), interchip communications (Core 1 of Processor 1 and Core 1 of Processor 2 in Node1) and finally, the slowest communication channel, which is internode (Any core from Node 1 with any core of Node 2).

These differences in designs and communication channels cause that parallel applications written using a message passing system (such as MPI [35]) vary in the behavior affected by the different latencies of the different available routes to send a message from one process to another.

2.1.2 Message Passing Interface

Parallel programming models are defined as an abstraction between the parallel hardware and the memory. These models allow programmers to easily develop parallel programs

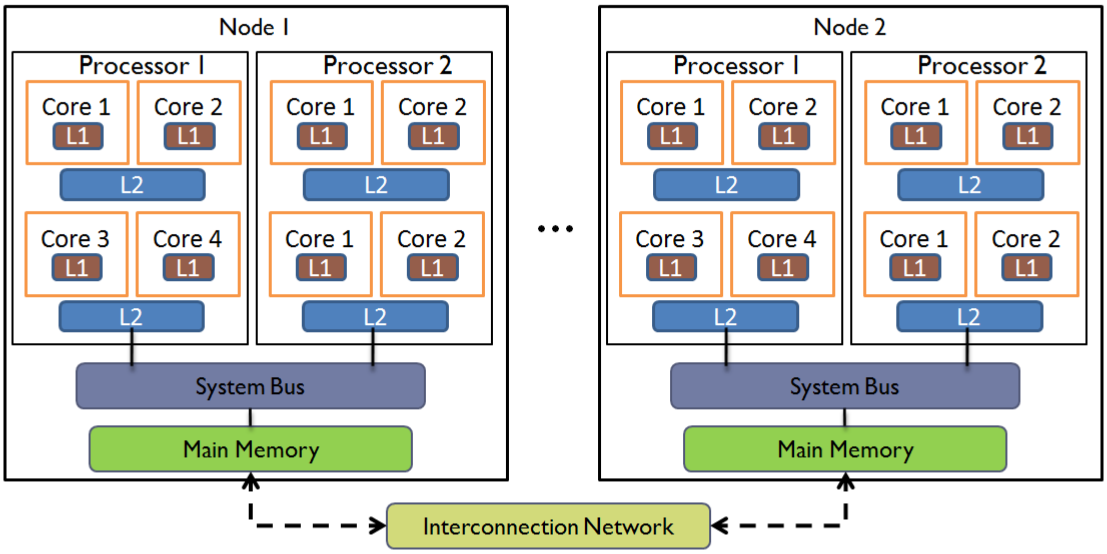


Figure 2.2: Quad Core Multicore Cluster.

that take advantage of the underlying architecture. There are two main models to write parallel applications: the shared memory model and the message passing model. In the shared memory model, processes can use the same memory sections, while in the message passing model the information between processes is transferred using messages.

In this thesis, we focus on the message passing model and we describe here the Message Passing Interface (MPI), which is the standard solution for programming in distributed memory systems.

The message passing model is used to develop efficient parallel programs executed in a distributed memory system. This model allows to control the information exchange between process of the parallel application. The main advantage of programs written using message passing libraries is that they can be executed using shared memory or distributed memory systems.

The *de facto* standard for the message passing model is the Message Passing Interface (MPI). In each message exchange an agreement between sender and receiver is made. Then, the sender process starts sending a message using the interface and the receiver process receives the message with some extra information. Messages in this model could be synchronous, asynchronous, blocking and non-blocking. There are also collective operations that make use of the basic operations and some other more complex functions to distribute and gather data.

The RADIC architecture and all the proposals made in this thesis have been included inside a MPI library called Open MPI [36]. This has been done in order to provide a transparent fault tolerant support to MPI applications that use this specific library.

Further details about the Open MPI implementation will be given in Section 2.3.

2.1.3 Main Parallel Metrics

There is a set of metrics that are taken into account when executing parallel applications in multicore clusters. In some cases, there is also a trade-off relationship between some metrics (eg. speedup and dependability), and one cannot be improved without making worse the other. We will describe in the next paragraphs the main metrics that are taken into account in this thesis:

- **Execution Time**: This is the main metric that we try to improve when parallelizing applications. In parallel executions, the execution time will be determined by the slowest parallel process, thus the workloads and overheads (when using fault tolerance) should be distributed homogeneously in order to avoid that the execution time of a process or a set of them is larger than others. It is important to highlight that fault tolerance tasks may affect negatively the execution time, but benefiting the dependability metric.

- **Speedup**: This metric is directly affected by the execution time. The speedup determines how fast a multi-processor environment solves an application. A simple approach to defining speedup is to let the same program run on a single processor, and on a parallel machine with p processors. The problem with this approach is that some parallel programs may be to large to fit on a single processor [28]. In this thesis, to calculate the speedup of a parallel application we take the best serial execution time and compare it against the parallel execution time [84]. The best serial execution time is obtained by multiplying the number of parallel process by the time that each process spend in computing its workload without taking into account communication times (Gustafson s model). In the ideal case we will obtain lineal speedup according to the number of processes executing the parallel application. However, one of the reasons that lower the speedup of parallel applications is the time spent in communications (overhead). This is why it is important to lower the impact on communications when using message logging techniques.

- **Load Balance**: This concept usually refers to the workload distribution among parallel processes. In a homogenous parallel machine, when the workload is equitably distributed among parallel processes we can say that the application is well-balanced. However, if we consider the impact of communications and memory we can talk about different boundaries that could affect applications performance. Executions where the performance is determined by characteristics of the processors are called *Computation-bound*. When the full capacity of the processor is not used, we usually have two main types of executions. Executions where the performance is determined by the memory subsystem are called *Memory-bound*, and when performance is determined by the communication times executions are *Communication-bound*.

- **Process Mapping**: This refers to the distribution or allocation of processes in a parallel machine. In order to make an efficient use of the parallel machine, it is very important to analyze the behavior of a parallel application searching for a suitable process mapping. For example, in order to minimize the number of network communications, processes with high communication volume could be placed in the same node. It is very common to use affinity in HPC in order to attach processes (process affinity) or threads (thread affinity) to a core in order to guarantee the re-usage of data or for load balancing issues. Processor affinity is a modification of the native central queue scheduling algorithm [70]. In this thesis we make use affinity to guarantee an homogenous distribution of load and to attach some fault tolerance tasks to specific cores.

- **Efficiency**: Good resource administration is a fundamental property that should be considered when executing in parallel environments. For this reason, when executing parallel applications we should focus on obtaining a balance between speedup and the percentage of time used per resources. Waiting times generated by communications or IO are translated into inefficiencies. The CPU efficiency index is defined as the time that processors are executing computation. Communication between processors is an important source of a loss of efficiency. If the load is balanced among the processors, they would always be performing useful work, and only be idle because of latency in communication. In practice, however, a processor may be idle because it is waiting for a message. Normally, efficiently values of parallel executions are between 60 and 90 percent [28]. In this thesis, we calculate the efficiency of a parallel execution by dividing serial execution time by the parallel execution time and by the number of processes used.

- **Scalability**: The performance of parallel applications depend on the problem size and the number of processes used to execute the application. The scalability metric refers to the adaptation level of the application to the parallel system. There are two main types of scalabilities: Weak and Strong. In weak scalability it is analyzed how the execution time varies when the problem size and the number for processing elements increase simultaneously, but the workload per process remains constant. The main objective in weak scalability is to maintain a constant execution time [71]. On the other hand, in strong scalability the problem size remains constant and the number of processing elements is increased (workload per process is reduced). In strong scalability the objective is to increase the speedup proportionally to the increase in the number of processes [65]. In scientific computing, scalability is a property of an algorithm and the way it is parallelized on an architecture. Taking this into consideration we have designed our fault tolerant architecture (RADIC) and our proposal to be scalable fault tolerant solutions.

- **Availability**: It represents the percentage of time that a system operates during its intended duty cycle. The availability value is affected according to the Mean Time To Failure (MTTF) and the Mean Time To Recover (MTTR) as shown in Equation 2.1. Taking into account that the MTTF usually depends on the parallel system configuration, in this thesis we focus on reducing the MTTR of failed processes in order to improve the availability.

$$Availability = MTTF \ \ (MTTF + MTTR) \tag{2.1}$$

- **Dependability**: It is a term that cover many system attributes such as reliability, availability, safety or security [30]. In this thesis we will focus on the availability attribute which refers to the percentage of time that a system operates during its intended duty cycle. The availability is improved by the use of fault tolerance techniques that allow systems to continue executing tasks even when losing some resources as consequence of failures.

- **Performability**: It is a composite measure of a systems performance and its dependability [64] [67]. It is the probability that a system performance remains above certain performance level during a specified period of time.

Many of the metrics that are considered to evaluate parallel execution refer to performance. However, as systems are growing in size and parallel applications are demanding more resources, the failure probability increases. Consequently, we cannot concentrate only in performance metrics but also in availability. In the next section we will cover the classic fault tolerance techniques used in HPC for improving availability that will help to frame this thesis.

2.2 Classic Fault Tolerance Mechanisms in HPC

Large parallel systems have vast computing potential. In these systems a machine can stop participating in execution of a parallel application as a result of disconnection from the network, shut down or power break. If any of these events occur we say that the node has failed. The computing potential is hampered by the nodes susceptibility to failures. It is necessary to preserve the correctness of a parallel execution despite failures. Therefore, fault tolerance is a valuable feature to provide high availability to computer clusters composed by multicore nodes.

During the last years, the failure rate in current systems has increased.In [79] has been conducted an analysis during nine years in Los Alamos Research Center. Between 40 % and 80 % of the incidences, depending on the cluster size, where caused by node problems.

Message passing applications are growing in number of processes in order to accomplish the requirements of more functionality, more data to handle and more performance. A node failure can have a dramatic effect on message passing application performance. As the quantity of faults in parallel applications has increased as consequence of failures, parallel applications should make use of fault tolerance techniques in order to finish their executions correctly.

In recent years many research has focused on the development of more efficient fault tolerance tolerance techniques appropriate to the new HPC architectures. The most used strategies are based on rollback-recovery, these strategies periodically use stable storage (e.g. disk) to save the processes states and maybe some additional useful data during failure-free execution. A saved state of a process is called a checkpoint. Upon a failure, restart a failed process from one of the saved checkpoints reduces the amount of lost computation, when recovering, consistency between processes must be maintained. Figure 2.3 depicts the main rollback-recovery protocols available currently (we highlight in bold the protocols that we use in this thesis and we include also our proposal).

Elnozahy [29] and Capello [18] warned that classical fault tolerance techniques such as coordinated checkpoint, will not be convenient in view of the growing size of systems. Elnozahy propose to save resources for the fault tolerance tasks and Capello indicates that more research should be done to reduce time consumption of fault tolerance tasks and to improve scalability of current solutions.

Several strategies have been developed to improve the performance of fault tolerance

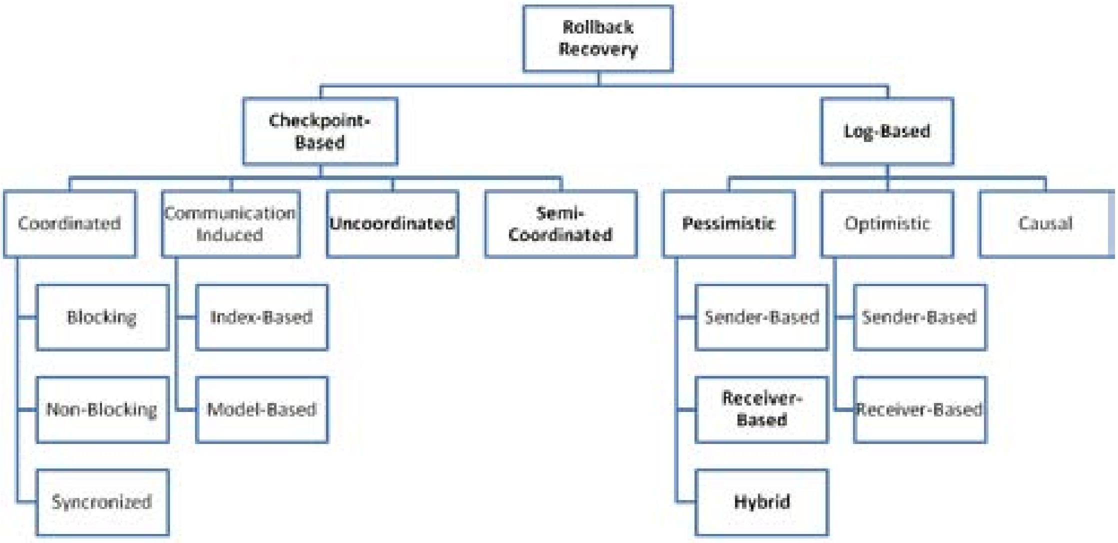


Figure 2.3: Rollback-Recovery Protocols.

solutions. In [21] was proposed a rollback-recovery protocol based on message-logging that focuses on lower the overheads in failure-free executions and avoids the restart of non-failed processes. Regarding checkpoint strategies, many improvements have been designed, such as: diskless checkpoint approaches [39], semi-coordinated checkpoint approaches [19] [13], techniques to determine the best checkpoint interval in uncoordinated approaches [34], among others.

Resource assignation and overhead management are two major aspects that should be cover in fault tolerance. In this thesis we focus on these two aspects, specifically when using message logging techniques.

2.2.1 Basic Concepts

We refer to a system as fault tolerant if it can mask the presence of faults in the system and try to provide the required services (probably degraded) by using redundancy at some level (hardware, software, temporal, information). Redundancy is the property of having more of a resource than is minimally necessary to do the job at hand (Koren and Krishna [50, Ch. 1]). When a failure affects a system, redundancy is exploited to mask or work around these failures in order to maintain a desired level of functionality.

In order to explain in detail the classical fault tolerance techniques and their operations we first define some basic fault tolerance concepts [27] that will be used in this thesis.

- **Fault**: is related to the notion of defect, incorrect step, process or data definition which causes a system or a component to perform in an unintended or unanticipated manner. Regarding their duration, faults can be *permanent*, when a faulty component goes down and cannot continue with its function until it is repaired or replaced; *transient*, when after a malfunctioning time the normal functionality of the component is restored; or *intermittent*, when the component swings between malfunctioning and normality. In this thesis we only focus on permanent faults that affect system s nodes.

- **Error**: is the manifestation of faults. Error is a discrepancy between a computed, observed or measured value or condition and the true, specified or theoretically correct value or condition. It is important to highlight that not all faults cause errors. We detect errors when a node is not reachable because of communication loss or inconsistent communication (when only some processes can reach the node).

- **Failure**:is the inability of a system or a component of it to perform its required functions or services. Fault tolerance techniques focus on avoiding that errors caused by faults become failures. In this work the failure system model is the next: when a process fails, the process loses its volatile state, stops its execution and does not send any more messages. Such behavior is called fail-stop. Processes have a stable storage device that survives failures and the fault tolerant system can sustain multiple failures of non-related components. The protocol selected do not tolerate failures during recovery in the components that are participating in the recovery. Our protocol guarantees that if the previous conditions are fulfilled, all processes will finish their execution.

- **Reliability**: represents the ability of a system or component to function under stated conditions for a specified period of time. The Mean Time Between Failure (MTBF) is the primary measure of a system s reliability.

- **Recovery line**: when an application is affected by a failure, the states of the affected processes are lost. The recovery line should be determined when rollback-recovery is being used, and it is the configuration of the entire parallel application after failed processes have been restarted from their checkpoints. As in parallel applications messages could cross between processes, a recovery set is the union of the saved states (including messages, events and processes states) and a recovery line [13].

- **Fault Masking**: fault masking techniques provide fault tolerance by ensuring that services are available to clients despite failure of a components. These techniques allow applications to continue their execution without knowing about hardware failures.

- **Stable Storage**: rollback-recovery uses stable storage to save checkpoints, event logs, and other recovery-related information despite failures. Stable storage in rollback recovery is only an abstraction, often confused with the disk storage used to implement it. There are different implementation styles of stable storage: In a system that tolerates only a single failure, stable storage may consist of the volatile memory of another node. In a system that wishes to tolerate an arbitrary number of transient failures, stable storage may consist of a local disk in each host.

- **Garbage Collection**: as the application progresses and more recovery information is collected, a subset of the stored recovery information may become useless. Deletion of such useless recovery information is called garbage collection. A common approach to garbage collection is to identify the recovery line and discard all data relating to events that occurred before that line. For example, processes that coordinate their checkpoints to form consistent states will always restart from the most recent checkpoint of each process, and so all previous checkpoints can be discarded. Garbage collection mechanisms in sender-based message logging are complex and could be costly since a checkpointed process should notify all the senders about its checkpoint so they can deleted old messages. In receiver-based approaches this is not the case, since the message log is located in one place and could be easily deleted.

- **Transparency**: general purpose fault tolerance approaches should be transparent to applications. Ideally, it should not require source code or application modifications. In parallel applications the most used option to provide transparent fault tolerance is the integration of techniques inside communication libraries [44] [59]. It is desirable to have architectures that could make an automatic fault management and that could carry out the protection, fault detection and recovery stages automatically.

- **General Purpose Fault Tolerance**: fault tolerance solutions must have a wide range of applications coverage. This reduces the software engineering and software development costs.

- **Portability**: fault tolerance techniques are preferred when they are not tightly coupled to one operating system version or to one specific application. A portable fault tolerance framework is desirable so it can be used in different systems.

Most of the current parallel applications are designed using MPI. The default behavior in parallel MPI applications is that if a single process is affected by a fault, the MPI environment notices this after some time and commands all remaining processes to also stop their executions (fail-stop) [35].

However, several strategies have been proposed in order to endow parallel executions with fault tolerance support. The most used strategy to do this is rollback-recovery, since it looks as the current best approach [29].

This protocols have been developed and compared over the years according to the overhead that they introduce during protection and recovery stages. During the protection stage, checkpoint and message logging techniques are used to save processes states and non-determinant events (messages). In the presence of hardware failures, the processes use the saved information to restart from a previous state and continue with their execution.

Following are detailed the main rollback-recovery protocols that were designed to provide fault tolerance for parallel applications.

2.2.2 Coordinated Checkpoint

Checkpoint-based approaches should be able to save Consistent System States. A global consistent state in a message-passing system is a collection of the individual states of all participating processes and of the states of the communication channels as well [29]. In a global consistent state, all messages receptions reflected by receivers should have a corresponding message emission from senders [22].

In coordinated checkpoint protocols is easy to determine a recovery line since all application processes save their states simultaneously [53]. In order to ensure that a consistent global state is saved, all in-transit messages are deleted when coordinating the processes.

Figure 2.4 depicts the coordinated checkpoint protocol behavior. Between moments A-B, C-D and E-F there might be communication between processes (not depicted in the figure). Between moments B-C and F-G the network is silenced to take snapshots of the parallel processes and produce a consistent global state. When the failure affects process Pn at instant D, all processes must rollback to instant C and restart over. The numbers in the figure represent the processes timeline, after the failure the processes need to repeat all computation made between instants C and D.

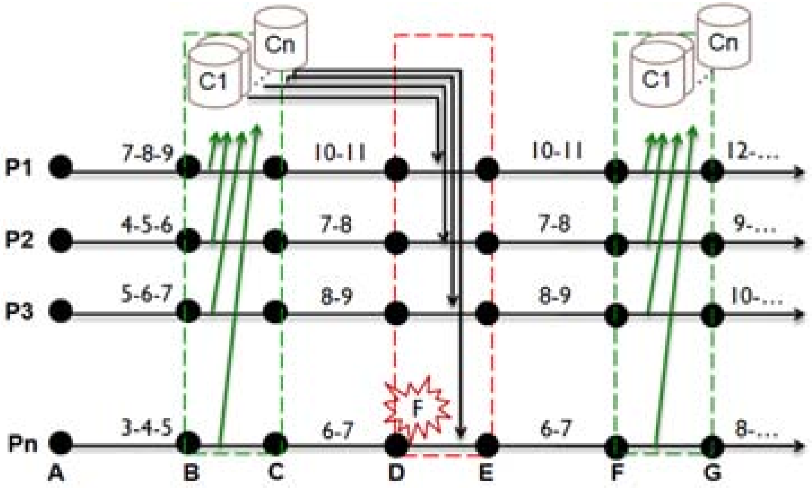


Figure 2.4: Parallel application running with Coordinated Checkpoint support.

The main drawbacks of coordinated checkpoint protocols are:

- When several processes are involved in a parallel execution, the coordination phase could take too much time.

- If some processes are smaller in size than others, their checkpoint time will be lower than the others but even if they finish checkpointing first they will wait for the others to finish. It is important also to consider that as application sizes increase, the global checkpoint size also increases and this could represent to the storage system.

- Storage devices and network could become a bottleneck when all the processes try to save their state at the same time.

- In case of a single process failure, all processes must rollback to a previous state losing all the computation done since the last checkpoint.

Despite these drawbacks, several fault tolerance protocols in parallel computing have implemented coordinated checkpoint mechanisms. In [44], Hursey describe in detail the design and implementation of a coordinated checkpoint approach inside the Open MPI library[36]. In this implementation, they propose to integrate the Berkeley Lab CheckpointRestart tool[41] inside Open MPI in order to provide checkpoint and restart support to parallel applications.

Ansel et al. in [2] describe the Distributed Multithreaded Checkpointing (DMTCP) tool. This tool can be used to provide coordinated checkpointing and restart to parallel and distributed applications.

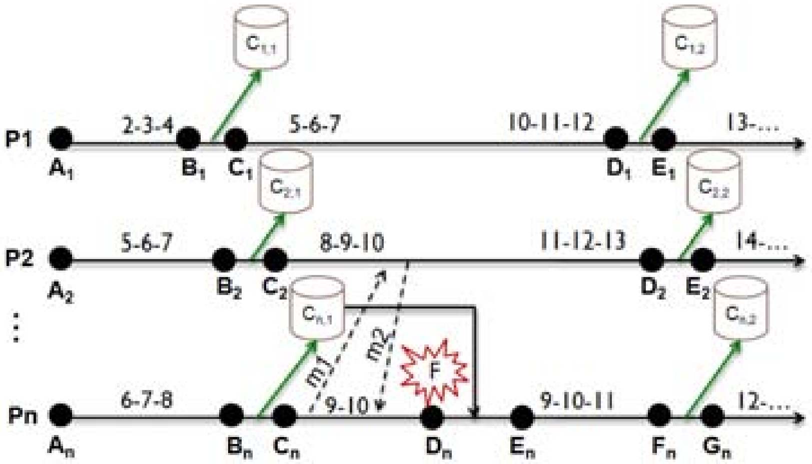


Figure 2.5: Parallel application running with Uncoordinated Checkpoint support

2.2.3 Uncoordinated Checkpoint

In uncoordinated checkpoint protocols each parallel process can select the most convenient moment to checkpoint its state, thus no coordination is needed.

In Figure 2.5 can be observed how the parallel processes take checkpoints without needing coordination ($B_1 - C_1$, $B_2 - C_2$, $B_n - C_n$), therefore no global checkpoint is created. In the default approach, each process takes its checkpoint independently and also keeps previous checkpoints. When a fault occurs, only the failed process rolls back to the last checkpoint ($D_n - E_n$). Non-failed processes can continue with their execution. In the ideal case, Pn will roll back to C_n and other processes will not be bother. However, if process Pn produces a message m1' instead of m1 when re-executing, m1 will be an orphan message, and if P2 did not send again m2, this will be a lost message. If P2 state depends on a non-deterministic event (receipt of a message m1) that cannot be reproduced, it is called an orphan process. Existence of orphan processes violates integrity of the execution and therefore must be prevented.

To determine a recovery line in an uncoordinated protocol is not as easy as in coordinated protocols. After a failure, non-failed processes may depend on the previous state of failed processes, thus some processes can be forced to roll back. If no message logging technique is used in combination with uncoordinated checkpoints, there is a high probability of creating orphan messages and losing messages. An orphan message is created when one process rolls back to a previous state before sending or receiving a message, but its counterpart does not roll back with it. A lost message is one that was not saved previously, thus the sender process should be forced to roll back (P2 will need to roll back to $C_{2\,1}$). This is the reason of storing more than one checkpoint version for each process. So if a process rolls back and a orphan message is created, then the counterpart should also roll back. Then, in order to avoid the domino effect [29], message logging techniques are used to create consistent recovery sets.

The main advantages of using uncoordinated checkpoints are related to the reduction in time when creating checkpoints and to the fact that, in ideal cases, only faulty processes must rollback. As no coordination is needed between processes, the time needed to perform a checkpoint tends to be smaller than in coordinated approaches. Then, no collective operation is needed, also storage and network bottlenecks could be avoided since checkpoints do not need to be saved simultaneously.

2.2.4 Message Logging

Message logging protocols are based on the assumption that process states can be reconstructed replaying all messages in the correct order [10]. Non-deterministic events (reception of messages, i/o operations, etc.) should be replayed in the same way they occur before the failure. Message logging techniques consider parallel applications as a sequence of deterministic events (computation) separated by non-deterministic events (messages) [51].

According to [52], message logging is expected to be more appropriated than coordinated checkpoint when the MTBF is greater than 9 hours. Furthermore, while coordinated checkpoint stop progressing when the MTBF is shorter than 3 hours, message logging techniques allow applications to continue progressing.

Before going any further, is very important to clarify some concepts that are used in message logging protocols [13] [56]:

- **In-transit Messages**: Let us consider the recovery line composed by $C_{1,2}$, $C_{2,2}$ and $C_{3,1}$ while using uncoordinated checkpoints in Figure 2.6. Messages m3 and m4 are crossing the recovery line and they are considered in transit messages. Supposing that P3 is affected by a failure, and it restarts from $C_{3,1}$, messages m3 and m4 will not be available anymore since P2 an P1 have already sent them in the past. In order to build a consistent recovery line, m3 and m4 should be saved as well as the checkpoints.

- **Lost Messages**: Messages m3 and m4 from Figure 2.6 could become lost messages during a re-execution since the send events are recorded but not the reception.

- **Delayed Messages**: These are messages that are sent by non-failed processes to failed processes that will arrive after the failed processes restart and reach the corresponding reception event. In Figure 2.6, m6 is a delayed message.

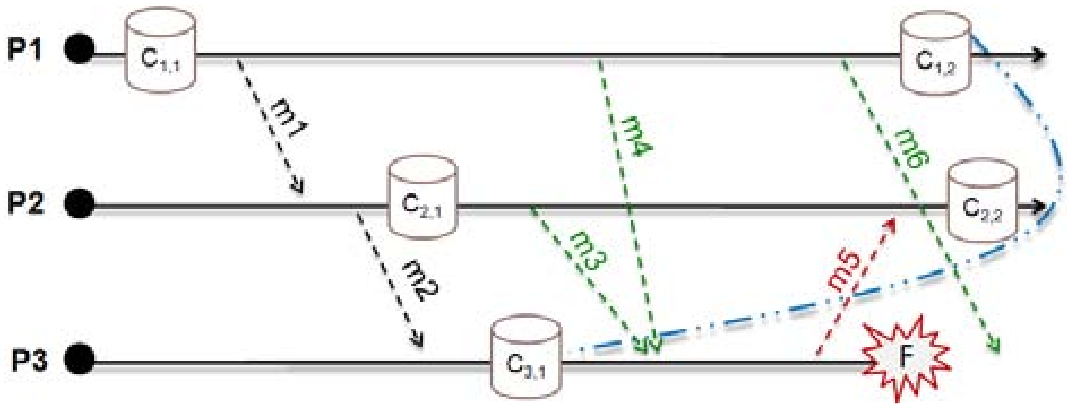


Figure 2.6: Message Types in uncoordinated checkpoint protocols.

- **Duplicated Messages**: These kind of messages are produced when a process rolls back and re-send previously sent messages. Considering Figure 2.6, if P3 fails, it will roll back to $C_{3\ 1}$ and send again m5 to P2. As m5 was already received by P2, this will be a duplicated message and it should be discarded.

- **Orphan Messages**: Considering the recovery line (Figure 2.6), message m5 is crossing it from a possible future of P3 to the past of process P2. Message m5 depends on messages m3 and m4 (taking into account Lamport's happened-before relationship [51]) and in a future re-execution the order of reception of these messages could vary (in MPI could be caused by the usage of a wild-card such as MPI_ANY_SOURCE). If this happens, instead of producing message m5, P2 could produce a different message (m5') and P1 will become an orphan process because is depending on a message that does not exist anymore.

Message logging techniques focus on providing a consistent recovery set from checkpoints taken at independent moments during the execution and they should be able to deal with the type of messages explained above.

Message logging is useful when the interactions with the outside world are frequent, because it permits a process to repeat its execution and be consistent without having to take expensive checkpoints before sending such messages [29]. Generally, log-based recovery is not susceptible to the domino effect, therefore they are a good complement to uncoordinated checkpoint techniques.

When using uncoordinated checkpoint protocols, message logging is used in order to build a consistent recovery set, all in-transit messages should be saved to avoid sender rollback, and non-deterministic events should be tracked to avoid the creation of orphan messages. Also duplicated messages should be discarded.

The main advantage of using message logging is that only failed processes must roll back, allowing other parallel processes to continue with their execution. While introducing some overhead on each message exchange, message logging techniques can sustain a much more adverse failure pattern, which is translated in better efficiency on systems where failures are frequent [52].

Message logging protocols can be classified in the next three types according to their way of operation:

- **Pessimistic**: In this kind of protocols the assumption is that the failure could occur immediately after the occurrence of a non-deterministic event, thus no process can depend on that event until it is correctly saved in stable storage. Each event is saved in a synchronous manner, thus increasing the overheads during failure-free execution but avoiding the generation of orphan messages. Pessimistic protocols guarantee that only failed processes rollback rollback to their last checkpoint, facilitate garbage collection and enable fast recovery.

- **Optimistic**: Each non-deterministic event is saved in an asynchronous way assuming that the events will be saved correctly before the occurrence of a failure. This kind of protocols reduce the overhead during failure-free executions, but orphan processes can be created.

- **Causal**: These protocols avoid the synchronous behavior of pessimistic approaches, but additional copies of messages are generated to avoid orphan messages. These approaches try to combine the advantages of the previous, but they maintain the drawbacks of optimistic protocols, since a more complex recovery algorithm is needed.

The mentioned message log protocols can be receiver-based or sender-based depending on where the message is saved. Receiver-based logging protocols normally doubles the message delivery latency during failure-free executions because every received message should be re-sent to a stable storage. Sender-based logging protocols introduce less overhead during failure free-executions but during recovery, sender processes should replay all messages exchanged with restarted processes. Garbage collection in sender-based protocols is more complex since extra communications between senders and receivers is needed to erase non-necessary messages. By contrast, for receiver-based message logging, all necessary data to rebuild the state of a restarted process is available in its log repository, therefore this protocol is the faster during recovery. Garbage collection in receiver-based approaches is less complex since after each checkpoint, receivers can delete their message logs.

Sender-based and receiver-based approaches are explained in detail in Section 4.1 since they are used as backbone in our Hybrid Message Pessimistic Logging proposal.

2.3 RADIC Architecture

In this section we describe the Redundant Array of Distributed and Independent Controllers (RADIC) [25] fault tolerance architecture. This architecture is explained in detail since it will be used to carry out the experimental validation of our proposals.

Adding scalable fault tolerance support to parallel applications may result in high software engineering and developing costs. An application transparent solution can highly minimize the costs in developing and testing. It is desirable to have fault tolerance solutions that handle all the necessary phases (Protection, Detection, Recovery and Reconfiguration). We describe in detail the RADIC architecture with all its characteristics, main components and operation modes. We describe how RADIC carry out automatically all phases allowing the configuration of a set of parameters in order adapt fault tolerance tasks to specific circumstances or necessities.

The RADIC architecture has been designed to cover 4 main functional phases, which are:

- Protection: RADIC uses rollback-recovery protocols that do not harm scalability of applications. An uncoordinated checkpoint approach in combination with a pessimistic receiver-based message logging approach are the default protocols of RADIC. The uncoordinated checkpoint guarantees that every process could take independent checkpoints. By combining this with a pessimistic logging strategy, we can ensure that in case of failure, only the affected processes must not rollback to a previous checkpoint. The overheads added during this phase depend on applications behavior and they could be higher than optimistic solutions. However, the protection decisions are taken in a decentralized manner at node level with neighboring information, then the scalability of our solution is not at risk.

- Detection: RADIC protects the parallel applications against node failures. These failures are detected by lack of communication of failed nodes. In order to guarantee a low failure detection latency RADIC uses a heartbeat/watchdog mechanism between different independent nodes to quickly find out about failures.

It is also possible to detect a failure by a faulty communication attempt. RADIC should be integrated in a software layer (e.g. the MPI library, socket-level) where it can observe and manage all message transmissions, so it can detect communication failures.

- Recovery and Reconfiguration: When a node failure occurs, RADIC starts the recovery phase by relaunching faulty processes from their latest checkpoints and after that, the processes consume the message log in order to reach the pre-failure state. Taking into account that the failed nodes will be removed from the application environment and processes will be restarted in a different location, the reconfiguration phase should deal with these changes. RADIC by default does not require extra components, but in order to avoid performance degradation spare nodes could be used to automatically replace failed nodes. As our approach does not rely on any central element, the reconfiguration is made on demand by using an algorithmic procedure to find the location of restarted processes when there are a communication attempts that involve them. These procedures are executed automatically, in order to reduce the MTTR to the sum of recovery time and re-execution time.
- Fault Masking: With the purpose of allowing a parallel application to continue its execution in presence of faults, we should avoid the error propagation to the application level. Thus, our layers capture the faults caused by the failure avoiding a fail-stop effect of the MPI application. Taking into account that processes migrate as a failure consequences, all communications with these processes should be redirected avoiding the application to notice about it.

2.3.1 Design Keys

Considering the main functional phases mentioned previously, the RADIC architecture has been designed taking into account the next design keys:

- Transparency: RADIC s main functionalities are designed to provide FT to parallel applications without modifying their source code. Its main features have been integrated below the application layer.
- Flexibility: Many features in RADIC could be set by parametric configuration to allow adaptation to the application needs or to different cluster configurations (E.g. Checkpoint Interval, activation of Pipeline Message Logging [78], fault tolerance

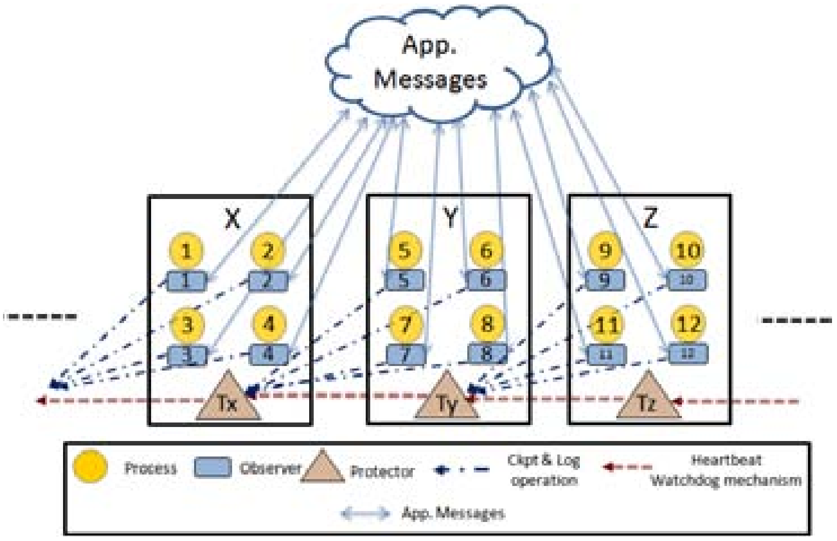


Figure 2.7: RADIC main components.

task mapping, number of spare nodes, etc.). According to the application needs, RADIC allows to define a protection degree (e.g. single node failure, multiple concurrent dependent failures).

- Automatic: RADIC performs its tasks of protection, detection and recovery automatically while the application is running, without needing any human intervention.
- Scalability: RADIC works in a completely distributed way and does not use any centralized component in order to avoid compromising applications' scalability. Moreover, all decisions taken during the execution are made in a decentralized manner using local data or by asking neighbor nodes, avoiding collective operations.

2.3.2 Components and Main Functions

RADIC provides fault tolerance support through two main components: Protectors and Observers. These components are initialized when the parallel application environment is created. On each node, one protector is running and each application process has one observer attached to it.

Protectors and observers work together with the aim of building a distributed fault tolerant controller and they have different functions to carry out. In the Figure 2.7 we illustrated three independent nodes (X, Y and Z), four processes are running per node and the main components of RADIC and how they interact with the application processes.

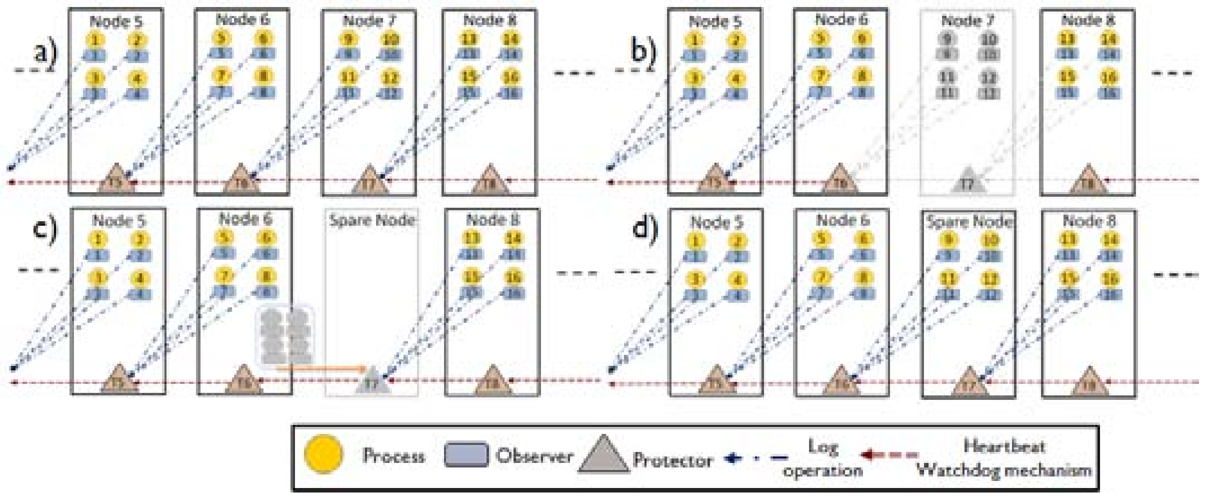


Figure 2.8: RADIC Scenarios: a) Fault free execution. b) Failure in Node 7. c) Utilization of spare Node , transference of checkpoints, Heartbeat/watchdog restoration and assignation of a new protector to processes of Node 8. d) Restart of faulty processes in Spare Node.

The main functions are explained below:

- Protectors: one protector is running on each node and their main function is to detect node failures via a heartbeat/watchdog protocol (detection mechanism). Protectors also store checkpoints and event logs (of protected processes) sent by observers (protection mechanism). When a failure occurs, the protector restarts the failed processes that it protects so they can re-execute from the last checkpoint. Protectors also has to reestablish the heartbeat/watchdog protocol since it gets broken due to node failures. Protectors collaborate with fault masking tasks, since while a process is restarting all incoming communications should be delayed and then the protector should provide the new location to processes that try to reach the restarted process. Protectors tasks need some computational and storage resources in order to manage the message logging protocol and to make checkpoints of processes.

- Observers: Each application process has one observer attached to it. Every communication is intercepted by the observers, so they have absolutely control of messages exchanged between peers. Observers are responsible of fault masking. During protection, the observers performs event logging of received messages in a pessimistic manner and periodically they take uncoordinated checkpoints of the process to which it is attached as well. The checkpoint could be also triggered by some events such as an end of a recovery or when the log-buffer is full. Checkpoints and logging data are sent and stored in their protectors located in another node.

Table 2.1: Main tasks of RADIC components.

Functional Phase	Protector	Observer
Protection	- Receive and save messages and checkpoints from protected processes. -Update its RADICTable.	- Send every received messages to protectors. - Make and send checkpoints to protectors
Detection	- Look for available processing unit to restart. - Monitor the correct operation of nodes by using Heartbeat/Watchdog. - Failure diagnosis (avoid false failures).	- Monitor communications and confirm failure with protector of failed process.
Fault Masking	- Tell observers to wait in communications attempts. - Communicate changes of RADICTable due to node failures.	- Intercept faults and delay pending communications until recovery is finished.
Recovery and Reconfiguration	- Transfer checkpoint and message log to the available node, if necessary. - Restart failed processes. - Update its RADICTable with the new data. - Reestablish Heartbeat/Watchdog mechanism. - Redirect observers to the new location of the process.	- Consume message log. - Take checkpoint after complete restart. - Find and set a new protector.

During recovery, the observers are in charge of delivering the message log to restarted processes until they reach the before-fault state. When processes finish their re-execution step, observers command a checkpoint. Observers also require some CPU cycles to manage message transmissions and also consume network bandwidth since received messages are forwarded to protectors.

It is important to mention that protectors and observers make use of a data structure called the *RADICTable*. This table has one entry by application process and each row contains: process id, URI (Uniform Resource Identifier), URI of process protector, the Receive Sequence Number (RSN) and Send Sequence Number (SSN).

The RADICTable is distributed among all protectors in the parallel environment and this table changes when failures occur, but it gets updated on demand by each protector (avoiding collective operations). The update process is explained in detail later.

2.3.3 RADIC Operation

RADIC protects a system by using their own computation resources, storing data of processes residing in a node (checkpoints and log of received messages) in a different node where failures are independent (Figure 2.8a). However, as RADIC does not require dedicated resources it consumes memory (buffers), storage, CPU and network bandwidth from the system when executing its fault tolerance tasks.

When failure occurs, the protector of failed processes restarts them inside its own node. However, this may overload the node and bring down the performance. In [59] we have proposed the inclusion and management of hot spare nodes inside the parallel execution environment in order to maintain the computational capacity even in presence of node failures, allowing us to maintain the initial configuration, this is called the Resilient Protection mode. In order to make use of the spare nodes, we have a data structure called SpareTable containing the Spare Id, the Address of the spare and the status (free or busy). The SpareTable is replicated among all protectors and its information is updated on demand as well as the RADICTables.

RADIC operations using the Resilient Protection mode can be observed on Figure 2.8. All message exchanges are made through Observers. After receiving a message, an Observer forwards it to a Protector residing in another node (Figure 2.8a). Protectors store checkpoints and message log of the processes that they protect. A node failure can be detected through a Heartbeat/Watchdog mechanism or by Observers that try to communicate with a failed process (Figure 2.8b). Failures are masked by delaying communications to faulty processes in order to avoid fail-stop behavior.

In order to maintain the computational capacity and the initial tuning of applications in presence of failures, failed processes will be automatically restarted in spare nodes [59] (Figure 2.8c). As can be observed in Figure 2.8d, after the recovery phase takes place, the initial configuration is maintained.

If we run out of spare nodes, failed processes are restarted in their protectors node, but performance may be impacted. This is called the Basic Protection mode, where performance degradation may occur when failures are present.

When a node fails and processes get restarted, the observers consult the RADICTable in order to find about the node where the process has been recovered by asking the process s protector. The protectors update the RADICTable on demand when they identify that processes have migrated because of failures.

The mechanism used to implement the dynamic redundancy at the resilient protection mode is very effective at keeping the same computational capacity avoiding node overloads. However, some stoppage may be necessary during preventive maintenance, e.g., replacing some fault-imminent nodes with healthy ones or replacing nodes by upgrade reasons.

RADIC allows the insertion of new spare nodes in the parallel environment while the application continues with its execution. The resilient protection mode also allows to perform preventive maintenance without needing to stop the entire application. This feature is based on the conjunction of a fault injector, and the mechanism described previously. This feature can be used to receive the process running in the fault-imminent node. With the failure injector, we can schedule the appropriate moment to inject a fault in referred node (just after taking its checkpoint, avoiding processing the event log). Hence, by using the spare mechanism previously described, the process running on the fault-imminent node will migrate to the new spare node added. Such a procedure allows the system administrator to replace cluster nodes without interrupt a running application. Some fault-prediction systems may also be used in conjunction with our solution, monitoring the node state, in order to trigger this mechanism when some values are reached.

Table 2.1 depicts the main functions of each component during the main functional phases. In case of failure, all communications with failed processes are delayed and then redirected to the new location of the processes.

2.3.4 Implementation Details

The first prototype of RADIC was called RADICMPI [26] and it has been developed as a small subset of the MPI standard. As a message passing library is very limited. As this implementation does not have all the MPI primitives, it cannot execute many of the scientific applications available.

RADICMPI does not consider collective operations and other complex functions that many applications use. For that reason, instead of extending the prototype to comply the MPI standard, we decided to integrate the RADIC architecture into a well-established MPI implementation. It allows the correct execution of any MPI application using the fault tolerance policies and mechanisms of RADIC. As shown in Figure 2.9, the fault tolerance layers of RADIC have been included beneath the application level and the MPI standard. This allows to provide transparent fault tolerance support without modifying the source code of applications and MPI functions.

In the next paragraphs we will explain some important features of the integration of RADIC into Open MPI.

Open MPI Architecture

A depth research about the inclusion of RADIC in Open MPI has been made in [33]. The implementation is named RADIC-OMPI and integrates the basic protection mode of RADIC. This initial implementation does not include spare nodes management, however the design and inclusion of Spare nodes into Open MPI is proposed in [59].

Open MPI architecture has been already described in [36]. For that reason, we will focus only on the components relevant to the RADIC integration.

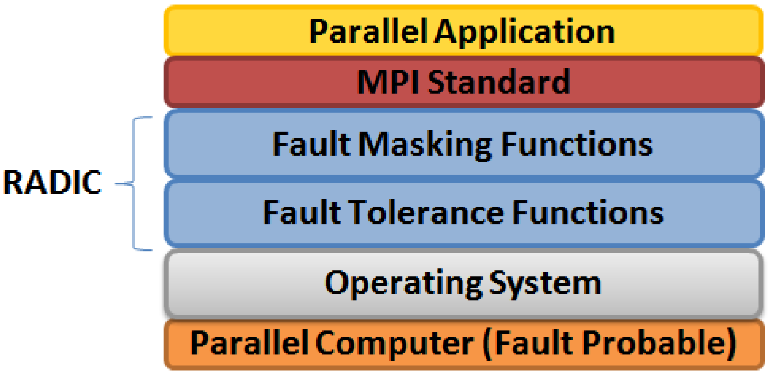


Figure 2.9: Fault Tolerance Layers.

The Open MPI frameworks are divided in three groups that are: *Open MPI (OMPI)* which provides the API to write parallel applications; *Open Run-Time Environment (ORTE)* which provides the execution environment for parallel applications; and *Open Portable Layer (OPAL)* which provides an abstraction to some operating system functions.

To launch a given parallel application, an ORTE daemon is launched in every node that takes part in the parallel application. These daemons communicate between them to create the parallel runtime environment. Once this environment is created the application processes are launched by these daemons. Every process exchange information about communication channels during the *Module Exchange (MODEX)* operation which is an all-to-all communication. The protector functionalities have been integrated into the ORTE daemon because in Open MPI one daemon is always running in each node, which fits the protector requirements.

OMPI provides a three-layer framework stack for MPI communication:

- *Point-to-point Management Layer (PML)* which allows wrapper stacking. The observer, because of its behavior, has been implemented as a PML component; this ensures the existence of one observer per application process.
- *Byte Transfer Layer (BTL)* that implements all the communication drivers.
- *BTL Management Layer (BML)* that acts as a container to the drivers implemented by the BTL framework.

The Open MPI implementation provides a framework to schedule checkpoint/restart requests. This framework is called *Snapshot Coordinator (SnapC)*. The generated checkpoints are transferred through the *File Manager (FileM)* framework. All these communications to schedule and manage the transferring of the checkpoint files are made using the *Out of Band (OOB)* framework.

RADIC inside Open MPI

To define the initial heartbeat/watchdog fault detection protection scheme and protection mapping a simple algorithm is used: each observer sets his protector as the next logical node, and the last node sets the first one as its protector.

All protectors should fill the RADICTable before launching the parallel application and update it with new information when failures are detected. The update of the RADICTable does not require any collective operation. Thus many protectors could have an outdated version of the RADICTable. However, the RADICTable will be updated further on demand, when observers try to contact restarted processes. All processes use the same initial data and apply the same deterministic algorithm to update the RADICTable.

Regarding to the fault tolerances mechanism and their integration into Open MPI, the following observations can be made:

- *Uncoordinated checkpoints:* each process performs its checkpoints through a checkpoint thread using the BLCR checkpoint tool [41]. Checkpoints are triggered by a timer (checkpoint interval) or by other events such as an specific message reception event. Before a checkpoint is made, to ensure that there are no in-transit messages, all communication channels are flushed and remain unused until the checkpointing operation finishes. Checkpoints are made using a component called *single_snapc*, which is a modification of the component provided by Open MPI in order to make independent checkpoints. After a checkpoint is made, each process transfers their checkpoint files using the *FileM* framework and then the communication within processes are allowed again.

- *Message Log:* since the observer is located in the PML framework (*vprotocol_receiver*), it ensures that all communications through it are logged and then transferred to the correspondent protector. Messages are marked as received by the remote process after the receiver and its protector confirm the message reception when using a pessimistic receiver-based log. When messages are sent and received, the SSN and RSN are updated accordingly. Considering the multicore environment and for giving more flexibility when distributing the message log tasks, we have implemented the logging managers of the protector as *Logger Threads*. Figure 2.10 shows a distribution of protectors tasks (without taking into account checkpointing), the logger threads are launched with as a different MPI application. Each application process that belongs to the real application gets connected to one logger thread (residing in its protector node) in order to save its messages.

- *Failure detection mechanism:* failures are detected when communications fails; this mechanism requires the modification of lower layers to raise errors to the PML framework where the faults are managed avoiding stops in the application execution. A heartbeat/watchdog mechanism is also used, where the protectors send heartbeats to the next logical node and the receiver protector resets the watchdog timer after reception.

- *Error management:* the default behavior of the library is to finalize when a failure occurs (fail-stop). Hence RADIC needs to mask errors to continue the executions and avoid fault propagation to the application level. When a protector finds out about a failure, the restarting operation is initiated. In order to manage the failures we have added a new *Error Manager* (*errmgr*) that allows the application to continue its executions if processes fail.

- *Recovery:* the recovery is composed of three phases. In the first one, a protector restarts the failed process from its checkpoint with its attached observer. Then the restored observer sets its new protector, re-executes the process while consuming the event logging data. When each reception operations goes through the *vprotocol_receiver* component during recovery, the messages are loaded from the log buffer. As messages are saved in a pessimistic manner, there are not orphan messages and messages sent by the restarted processes are discarded by receiver processes according the SSN value. Protectors involved in the fault also reestablish the protection mechanism. We consider the recovery as an atomic procedure.

- *Reconfiguration:* when the recovery ends, the communications have to be restored. To achieve this goal the lower layers of Open MPI must be modified to redirect all the communications to the new address of the process. To avoid collective operations this information is updated on demand or by a token mechanism.

The main problem when restarting a process in another node is that we need an ORTE daemon running in that node to adopt the new process as a child. Moreover, all future communications with the restarted process needs to be redirected to its new location. For that reason, ORTE daemons are launched even in spare nodes, but no application process is launched on it.

An additional problem that must be addressed is that a sender observers must not consider as a failure the lack of communication with other processes when receiver processes are doing a checkpoint or restarting. When a sender observer fail to communicate, it will

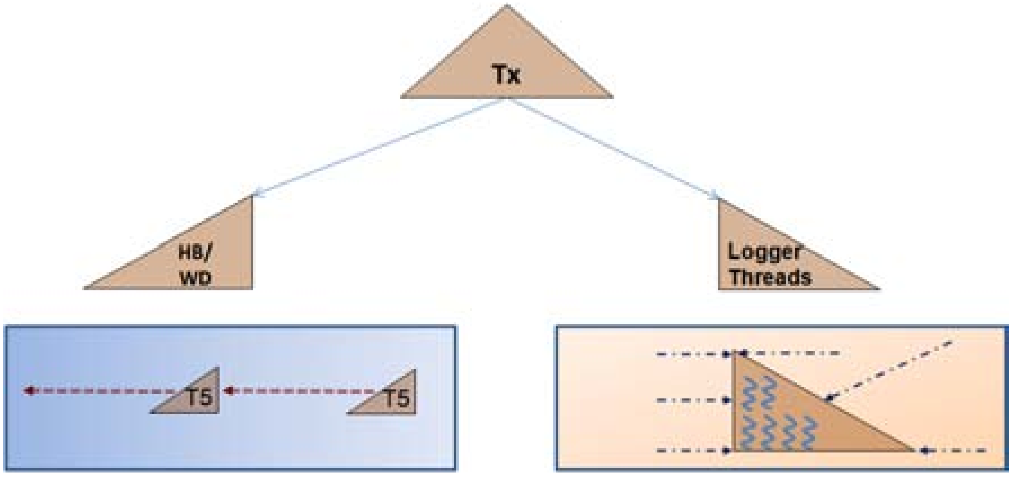


Figure 2.10: Dividing Protector Tasks.

consult the receiver s protector to find about the state of the receiver. The protector will indicate that the process is checkpointing or restarting, and the communication will be retried later.

The RADICTable and Sparetable were included inside the job information structure (*orte_jmap_t*). When the parallel application starts, each protector (*ORTE daemon*) populates its RADICTable and its Sparetable. The RADICTable and Sparetable are updated (on demand) when a protector notices that a process has restarted in another place. If the application runs out of spares, the basic protection mode of RADIC is used.

Chapter 3

Related Work

In this chapter we present the related work of this thesis. Taking into account that we focus on performance improvement while using fault tolerance techniques, we have divided this chapter in three main sections which are:

- Section 3.1 describes studies that focus on improving performance of applications executed in multicore clusters.
- Section 3.2 covers studies and optimizations developed in the message logging field.
- Section 3.3 describes solutions composed by a combination of fault tolerance techniques and discuss their relationship with the RADIC architecture used in this thesis.

3.1 Performance Improvement

Several studies regarding performance improvement of parallel applications running on multicore clusters have been made. In [55], Liebrock presented a method that allows programmers to execute applications in hybrid parallel systems with the aim of improving adaptability, scalability and fidelity. In order to reach its objective, this work proposes to apply mapping techniques according to the parallel environment characteristics.

Other study that focuses on improving performance metrics was presented by Vikram in [83]. In this work was presented a strategy to map tasks in reconfigurable parallel architectures. The aim is to obtain the maximum possible speedup, for a given reconfiguration time, bus speed, and computation speed.

The work presented in [57] focuses on efficiently placing MPI processes according to a policy for improving performance. Then, in [65] a method that attempts to enhance the performance of SPMD applications by combining scalability and efficiency was presented. The goal of this method is to obtain a combination between the maximum speedup under a defined threshold of CPU efficiency. This work has been used as basis when we focus on the case study of executing SPMD applications efficiently with fault tolerance support.

In [45] was presented an algorithm for distributing processes of parallel applications across processing resources paying attention to on-node hardware topologies and memory locality. Regarding the combination between mapping of fault tolerance tasks, specifically message logging tasks, and application process mapping, to date, no works have been published to the best of our knowledge.

3.2 Message Logging

Several works have been developed to improve the performance and minimize the overhead of message logging protocols. In [86] was presented a sender-based message logging approach to avoid sympathetic rollback of non-failed processes. Sympathetic rollback means that a process rolls back to re-send a message that has been lost because the receiver has rolled back to a previous state. This paper does not address the problem of rolling back because of orphan messages creations. They propose to detect messages that can never cross a possible recovery line.

In [73] is presented an extensive analysis of message logging protocols and a comparison of sender-based, receiver-based and causal protocols is performed. According to the obtained results, they conclude that is not the best option to rely on other processes to provide messages to restarted processes. Sender-based and causal protocols incur in high recovery costs because of this. They also recommend that optimistic protocols should focus on developing orphan-detection protocols, instead of only focusing on performance metrics.

In [52] is compared a coordinated checkpoint protocol with a pessimistic sender-based message logging implemented in MPICH-V [9]. The obtained results demonstrate that message logging approaches can sustain a more adverse failure pattern than coordinated checkpoints.

In [11] a comparison between a pessimistic and optimistic sender-based message logging approaches is presented, and both seem to have a comparable performance. Nevertheless, when using sender-based protocols and a failure occurs, processes that were not involved in the failure may need to re-send messages to restarted processes.

In [12], they propose to reduce the failure-free overheads in pessimistic sender-based approaches by removing useless memory copies and reducing the number of logged events. Nevertheless, the latency and complexity in the recovery phase are increased because of the usage of sender-based protocols.

In [78] was proposed a mechanism to reduce the overhead added using the pessimistic receiver-based message logging of RADIC. The technique consists in dividing messages into smaller pieces, so receptors can overlap receiving pieces with the message logging mechanism. This technique and all the RADIC Architecture has been introduced into Open MPI in order to support message passing applications.

However, it is important to highlight that the performance of a fault tolerance technique will depend on the target system and application characteristics as well as the fault tolerance parameters used. In [80] is proposed a method to automatically select a suitable checkpoint and recovery protocol for a given distributed application running on a specific system.

3.3 Fault Tolerance Solutions

In this section, we first describe protocols that combine classic fault tolerance techniques seeking to improve the performance of these approaches. Next, we describe other fault tolerant solutions that have been developed during the last years.

3.3.1 Hybrid Protocols

Rollback-recovery protocols need to evolve as well as HPC systems also evolve. In recent years, many works have focused on designing new protocols (or combination of old ones) to improve availability while reducing overheads and recovery times.

Classical pessimistic message logging protocols propose to save messages in a synchronous manner during failure-free execution, thus introducing high overheads in messages that are transferred between processes inside the same multicore node. Depending on where the messages should be logged (e.g. a different node), the latency of an internode message may be added to the latency of each intercore message.

Coordinated and centralized checkpoints avoid the overhead of message logging techniques, but on the other hand the coordination overheads and checkpoint saving time may increase considerably when the number of processes increases as well.

Works like [43], [56] and [76] focus on grouping the processes that communicate more frequently in order to reduce the number of messages logged using a coordinated checkpoint between these processes.

HOPE [56] is a hybrid protocol that combines Communication Induced Checkpoints (CIC) [29] with a pessimistic logging protocol. This work considers Grid environments, and the grouping of processes is done according to the network and communication pattern. The CIC is not efficient neither scalable between tightly coupled applications, since the number of checkpoints could increase in an uncontrolled way.

In [37], Gao proposes to create coordinated groups and overlap the checkpoint operation between each other. This work focuses on reducing storage bottlenecks by scheduling checkpoints at different times. They propose to avoid the usage of message logging by using a coordination protocol. However, in case of failure all processes should rollback to the last global checkpoint.

Other works such as [19] and [13] consider that the main unit of failure in a computer cluster is a node. Therefore, they focus on developing techniques to provide a hybrid or semi-coordinated checkpoint approaches, where a coordinated checkpoint is used inside the node and messages between nodes are logged to stable storage.

Considering message logging approaches, in [73] besides the extensive analysis of classic message logging protocols, they also describe hybrid message logging protocols (Sender-based and Causal). They propose an orphan-free protocol that try to maintain performance of sender-based protocols while approaching the performance of receiver-based protocols. The sender in this case synchronously save messages and the receiver asynchronously save them into stable storage. This proposal is similar to our Hybrid Message Pessimistic Logging (HM_{PL}) approach, however further details about the design is missing. They do not cover the mechanisms to avoid orphan processes creation and do not specify how and where the received messages are stored. On the other hand, our proposal is based on a distributed stable storage where each process saves messages on a different node (distributed storage) and orphan processes are avoided by detecting source of non-determinism while receiving messages.

3.3.2 Other Fault Tolerance Solutions

There is several ongoing research on fault tolerance architectures or complete fault tolerance solutions in message-passing parallel systems. Most of them, like RADIC, are based on rollback-recovery strategies, because they have proved to be the more suitable for the majority of parallel applications.

In this section, we have selected well-known research projects from the literature and we relate them with RADIC in characteristics that are useful to assess different proposals. These characteristics are summarize in Table 3.1 and explained as follows.

The *Message Logging* column specifies the types of message logging support of the

Solution	Message Logging	Fully Decentralized	Checkpoint	Storage	Transparency	Automatic Det. -Recovery	Warm Spares
RADIC	Pessimistic Receiver	Yes	Uncoord. or Semi-Coord.	Distributed	Yes	Yes	Dynamic
OPEN-MPI	No	No	Coordinated	Central	Yes	No	N/A
MPICH-V	Several Strategies	No	Uncoord. or Coord.	Central	Yes	No	Pre-defined
FT-Pro	No	No	Coordinated	N/A	Yes	Yes	Pre-defined
MPI/FT	No	No	Coordinated	Central	No	Yes	Pre-defined
LAM/MPI	No	No	Coordinated	N/A	Yes	No	N/A
FT-MPI	No	No	N/A	N/A	No	No	Pre-defined
Starfish	No	No	Uncoord. or Coord.	N/A	Yes	Yes	N/A

Table 3.1: Fault tolerance in message passing parallel systems

solution. The *Fully Decentralized* feature means the components and functions used to tolerate failures are distributed and the decisions are decentralized. Consequently, the fault tolerance functional phases, including recovery, no central elements are used to take global decisions neither collectives instructions are performed.

The *Checkpoint* column indicates which are the checkpoint mechanisms supported in each case. One of the main problems of coordinated checkpoint is that when increasing the number of processes involved in the coordination step, the necessary time to coordinate the processes becomes important.

The column *storage* is used to show if the solution uses central or distributed storage for saving redundant data such checkpoint or message log.

The *transparency* is achieved when no changes has to be done to the parallel application code to use a fault tolerance solution.

The feature of *Automatic failure detection and recovery* allows the application to continue its execution in case of failure without needing administrator intervention or programming additional scripts for these tasks. This feature helps to increase the availability of the system by decreasing the repairing time.

Finally, we have analyzed the presence of *warm spare* feature, considering as *dynamic* when allows a dynamic insertion of spare node during execution and *pre-defined* when an application starts with a predetermined number of spares being used as fault occurs. RADIC enables the usage of spare nodes in case of failure but these mechanisms also may be applied to perform preventive maintenance by making a hot swap of machines. Other solutions use spare nodes as well or allow preventive activities.

The MPI 3 fault-tolerance working group try to provide fault-tolerance features to the applications in order to support algorithm-based fault tolerance. They propose a distributed consensus that may be used to collectively decide on a set of failed processes. In [16] is described a scalable and distributed consensus algorithm that is used to support new MPI fault-tolerance features proposed by the MPI 3 Forum s fault-tolerance working group.

Open MPI has an infrastructure to support checkpoint/restart fault tolerance [44] by providing a transparent interface that allows users to use fault tolerance techniques when executing MPI parallel applications. Although it is intended to be extended to other rollback-recovery protocols, this paper presents an implementation of coordinated checkpoint/restart using a snapshot coordinator that makes centralized decisions. System operators or programmers are responsible for application restarts.

MPICH-V [9] is a framework composed by a communication library based on MPICH and a runtime environment. MPICH-V is a complex environment involving several entities: Dispatcher, Channel Memories (CM), Checkpoint servers, and Computing/Communicating nodes. Channel Memories are dedicated nodes providing a service of tunneling and repository. MPICH-V1 [7] proposes a receiver-based protocol using CMs (dedicated nodes) to store received messages. Checkpoints are done independently but saved in a checkpoint server. The CMs and the checkpoint server can become a bottleneck when several processes are used.

In [8], Bouteiller et al. presented MPICH-V2. They proposed to use a sender-based message logging to reduce message latencies problems of MPICH-V1. However, the use Event Loggers in dedicated nodes can produce bottlenecks problems, and the latency for short messages is increased since before sending a message each process should wait for a confirmation of the last event saved in the event logger. The re-executions are slower than in MPICH-V1 since a restarted process should first contact its event logger and then asks for logged messages. MPICH-Vcausal [9] focuses on reducing the latency cost and the penalties during re-execution by using a causal message logging. In [14] they proposed a Chandy-Lamport based coordinated checkpoint which focuses on lowering the stress during recovery in the storage servers by saving checkpoints in local nodes.

Comparing to RADIC, the main difference between MPICH-V framework and all its features is that these approaches use centralized checkpoint servers to store the redundant data, and also dedicated resources to store message determinants. During the recovery process MPICH-V can use spare nodes, but a facility for dynamic spares insertion is not mentioned.

FT-Pro [54] is a fault tolerance solution that is based on a combination of rollback-recovery and failure prediction that allows to take some action at each decision point.

This approach aims to keep the system performance avoiding excessive checkpoints. Three different preventive actions are currently supported: Process migration, coordinated checkpoint using central checkpoint storages or no action at all. Each preventive action is selected dynamically in an adaptive way intending to reduce the overhead of fault tolerance. FT-Pro only works with an static number of spare nodes and it demands a central stable storage which differs from RADIC s approach.

MPI/FT [4], in opposite to RADIC approach, is a non-transparent solution to provide fault tolerance for MPI applications. The strategy is based on middleware services specifically tailored to meet the requirements of the application model and the communication topology. Two programming style are considered. Firstly, Master-Slave with a star communication topology which only need checkpoint/restart for the master process. Secondly, Regular-SPMD with all-to-all communications which is checkpointed and restarted in coordinated way suitable for its synchronous behavior. The recover procedure is based on an interaction of two elements, a central coordinator and self-checking-threads (SCTs) that use spare nodes taken from a pool. As this solution does not allow dynamic insertion of new spares, the application will fail after the exhaustion of this pool. When a failure is detected, the application is warned so it is in charge of performing the recovery.

The LAM/MPI [77] implementation uses a component architecture called System Services Interface (SSI) that allows the usage of a checkpoint library, such as BLCR, to save the global state of an MPI application using a coordinated checkpoint approach. This feature is not automatic, needing a back-end restart from the administrator. In case of failure, all applications nodes stop and a restart command is needed. Unlike RADIC, this procedure is neither automatic, nor transparent.

FT-MPI [31] has been developed in the frame of the HARNESS [5] metacomputing framework. The goal of FT-MPI is to offer to the end user a communication library providing a MPI API, which benefits from the fault-tolerance in the HARNESS system. FT-MPI presents the notion of two classes of participating processes within the recovery: Leaders and Peons. The leader is responsible for synchronization, initiating the Recovery Phase, building and disseminating the new state atomically. The peons just follow orders from the leaders. If a peon dies during the recovery, the leaders will restart the recovery algorithm. If the leader dies, the peons will enter an election controlled by a name service using an atomic test and set primitive. A new leader will restart the recovery algorithm. This process will continue either until the algorithm succeeds or until everyone dies. Such a solution is not transparent to the programmer, needing some user code to deal with faults and to start the recovery process.

Starfish [1] provides failure detection and recovery at the runtime level for dynamic and static MPI-2 programs. This architecture is based on daemons running on each node which forms the Starfish parallel environment. Starfish provides system initiated checkpointing, automatic restart and a user level API which can be optionally used at application level to control checkpoint and recovery. Both coordinated and uncoordinated checkpoints strategies may be applied by the user choice. For an uncoordinated checkpoint, the environment sends to all surviving processes a notification of the failure and the application may take decision and corrective operations to continue execution. As there is no event log, the recovery process using uncoordinated checkpoints may lead to domino effect. Starfish requires a central element called management module, which may affect its scalability.

All of the solutions presented above differs in some way from RADIC features. The Table 3.1 summarize these differences by comparing with the six features previously explained.

In works like [74] and [75], portable and transparent checkpoint-based protocols are presented. The CPPC framework, detailed in [74], focuses on providing an OS-independent checkpoint solution for heterogeneous machines. The CPPC library provides routines for variable level checkpointing and the CPPC compilers helps to achieve transparency by adding fault tolerance instrumentation code during the compilation phase. Such ideas could be adapted to be used inside the RADIC architecture in order to reduce checkpoint sizes, thus benefiting the checkpoint transference time and decreasing the storage size used.

During the last years, great effort has been put in order to add fault tolerance management to the MPI standard [42] but it is still an open issue. These kind of proposals are intended to change the most used fail-stop behavior by adding MPI primitives in order to give awareness of a failure to the application level. After learning about a node failure, the application is able to start the recovery procedure using the available process state knowledge.

Bland et al. [6] describes a set of extended MPI routines and definitions called user-level failure mitigation (ULFM), that permits MPI applications to continue communicating across failures. ULFM was proposed as an extension to introduce fault-tolerance constructs into the MPI standard.

Hursey et al. [45] propose failure handling mechanisms naming run-through stabilization. Many new constructs to be added to the MPI standard are introduced with the aim of making a failure being recognized by the application and allowing to recover and continue using the communicators.

Both proposals are not transparent to the application and do not provide a mechanism for automatic detection and recovery. Consequently, the cost of developing and testing for existing applications would be high. Nevertheless, they would open the possibility to provide a transparent fault tolerance layer at application level that today is not possible because in the current MPI standard the communications are entirely controlled by the MPI libraries implementations. However, RADIC is conceived to work properly with new versions of MPI libraries assuming that only one fault tolerance controller is activated during the execution to avoid duplicated tasks.

Taking into account the current trend of executing computational intensive application in cloud environments and the failure rate of these environments, fault tolerance would be a necessary feature. In [38] is proposed a fault tolerant virtual cluster architecture that focuses on restoring the application performance when one part of a distributed cluster is lost. As a future work, we are planning to adapt RADIC fault tolerance policies to cloud environments in order allow applications to comply with their deadlines when failures isolate one part of the execution system.

Chapter 4

Improving Current Pessimistic Message Logging Protocols

In this chapter, we discuss in detail the most used message logging techniques to provide fault tolerance support in uncoordinated approaches. We also present our proposed new message logging approach, which focuses on combining the advantages of receiver and sender based approaches, this new technique is called Hybrid Message Pessimistic Logging (HM_{PL}).

Uncoordinated fault tolerance protocols, such as message logging, seem to be the best option for failure prone environments since coordinated protocols present a costly recovery procedure which involves the rollback of all processes. Pessimistic log protocols ensure that all messages received by a process are first logged by this process before it causally influences the rest of the system.

In this chapter we describe in detail the pessimistic version of receiver-based message logging (RBML) and also the pessimistic version of the sender-based message logging (SBML). We focus on pessimistic versions because they ensure that in case of a failure there is no need to rollback non-failed processes.

Taking into account that most of the overhead during failure-free executions is caused by message logging approaches, the Hybrid Message Pessimistic Logging focuses on combining the fast recovery feature of pessimistic RBML with the low protection overhead introduced by pessimistic SBML.

The key concepts, design and main features of the HM_{PL} are explained in this chapter. The protection and recovery mechanisms are also discussed and validated by integrating the HM_{PL} inside the RADIC fault tolerant architecture.

The message logging techniques that will be explained here and in the rest of this thesis consider the utilization of an MPI library. There is no problem to extend the concepts presented here to other communication libraries or layers.

4.1 Message Logging Description

The main drawbacks of coordinated checkpoint approaches are the synchronization cost before each checkpoint, the checkpoint cost and the restart cost after a fault, since all processes should rollback. Message logging techniques do not suffer from these problems, since processes checkpoint and restart independently. However, the log adds a significant penalty for all message transfers even if no failures occurs during the execution [14]. According to these differences, the best approach depends on the fault frequency, process number, communication pattern, processes synchronization, among others.

It is important to highlight that log-based protocols combine checkpointing with message logging in order to enable a system to recover from the most recently checkpoint. These protocols are based on the assumption that the computational state of a process is fully determined by the sequence of received messages (Piecewise Deterministic) [29]. The sufficient condition to define a consistent global state, from where a recovery can be successful, is that a process must never depend on an unlogged non-deterministic event from another process.

Message logging techniques allow applications to sustain a much more adverse failure pattern than coordinated approaches, mainly due to a faster failure recovery [52]. When using a message logging technique we can allow parallel processes to take checkpoints independently and in case of failure non-failed process can continue with their executions if they do not depend on one or more failed processes.

As message logging techniques could allow applications to restart faster, this will be translated into better efficiency on systems where failures are frequent. On the other hand, there is an added overhead that could be considerable in some cases since every message should be treated as potentially in-transit or orphan. As every outgoing or incoming message should be saved (depending on the strategy), there is an extra time that is added to each message transmission.

Pessimistic log protocols ensure that all messages received by a process are first logged by this process before it causally influences the rest of the system, so they avoid the creation of orphan processes. In this section, we focus on describing the most used and implemented pessimistic message logging techniques, their main advantages and limitations. We will present the details of the receiver-based logging approach and the sender-based logging approach which are used as pillars of the HM_{PL}.

The HM_{PL}, which is presented in the next section, focuses on reducing the overheads

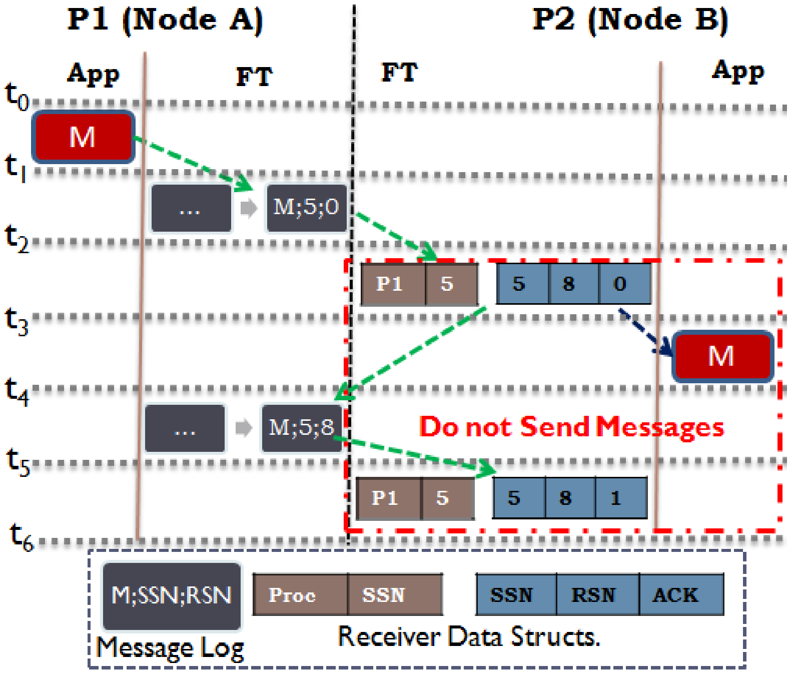


Figure 4.1: Sender-Based Message Logging.

in each message transmission caused by receiver-based approaches while maintaining the fast-recovery feature of this approach. Thus, it tries to conceal the advantages of sender-based and receiver-based message logging.

4.1.1 Sender-Based Message Logging

The SBML [48] is a solution that focuses on introducing low overhead during failure-free executions. Non-deterministic events are logged in the volatile memory of the machine from which the message is going to be sent. The main idea behind this message logging technique is to avoid the introduction of high overheads in communications and delays in computations by asynchronously write the messages to stable storage.

In Figure 4.1 we illustrate the operation of a pessimistic version of the SBML with the main data structures that it requires. The logical times (t_x) that can be observed at the left of the figure are there only to indicate precedence of steps. We have splitted each process tasks into Application tasks (APP) and fault tolerance (FT) tasks.

The main data structures and values used in the SBML are:

- *Send Sequence Number (SSN)*: this value indicates the number of messages sent by a process. This value is used for duplicate message suppression during the recovery phase. When a process fails and recovers from a checkpoint, it will re-send some messages to some receivers. If the receiver processes has the current SSN value for that sender, they will be able to discard these duplicated messages.

- *Receive Sequence Number (RSN)*: this value indicates the number of messages received by a process. The RSN is incremented with each message reception and then this value is appended to the ACK and send back to the sender.
- *Message Log*: this is where the sender saves each outgoing message. Along with the payload of the message also the identification of the destination process and the SSN used for that message are saved. When the RSN of the message is returned by the receiver, it is also added to the log.
- *Table of SSN*: this table is located in the receivers and it registers the highest SSN value received for messages of each process. The information saved in this table is used for duplicate message detection.
- *Table of RSN*: this table is located in the receivers and has one entry by each received message. It is indexed by SSN and contains the RSN and a value that indicates if the ACK of the of the message has been received by the sender.

When a process is checkpointed all these data structures are saved also, except the *Table of RSN*, since all received messages now are part of the checkpoint. All messages sent to an already checkpointed process should be deleted from the message log.

In Figure 4.1 we assume that the message M includes the headers with information about sender, destination and also the payload of the message. When the message M is about to be sent (from P1 to P2), P1 first save the message M and the SSN.

Once the message is saved into a buffer of the sender, the message M and the SSN are sent to P2. P2 saves the sender ID (P1) and the SSN in the *Table of SSN*, also it increments its RSN. After this, P2 add this RSN to the *Table of RSN* and send and ACK with the current RSN to P1. P1 receives and adds the RSN to the *Message Log*, then P1 sends an ACK that P2 receives and adds to the *Table of RSN*.

There is almost no delay in computations while the message logging is taking place since P1 can continue its execution after saving the message and P2 after receiving it. Nevertheless, between the reception of a message and the ACK with the RSN included, the receiver process should not send messages [48].

It is possible that processes fail while some messages do not yet have their RSNs recorded at the sender, these messages are called partially logged messages. If P2 fails and return from a checkpoint it will broadcast requests for its logged messages and the fully logged messages will be replayed in ascending RSN order, starting with the stored RSN+1.

Partially logged messages will be send in any order. As receiver processes cannot send messages while a message is partially logged, no other processes than P2 can be affected by the receipt of a message that is partially logged.

If we consider that P1 fails and retransmits M with the SSN equal to 5, P2 will discard it according to the current SSN value, and if the ACK of this message was not received by P1, P2 will send it.

However, if a process rolls back to a previous state, it will ask all the senders for its logged messages. Thus, the senders would have to stop their executions and look for these messages, unless a FT thread is in charge of managing the message log.

An approximation of the overheads of the protection stage in the a pessimistic SBML approach is represented in Equation 4.1, Equation 4.2 and Equation 4.3, where:

- T_Sender is the time spent by the sender when logging the message.
- $Wait\_ACK$ is the time spent by the process waiting for an ACK.
- $T\_Receiver$ represents the time spent by the receiver when receiving the message.
- I_{Log} and U_{Log} represents the cost of inserting and updating the message log respectively.
- $I_{DataStructs}$ and $U_{DataStructs}$ represent the cost of inserting and updating the data structs in the receiver process. $I_{DataStructs}$ includes the time spent in inserting a new element in the Table of SSN and Table of RSN. $U_{DataStructs}$ represents the time spent in updating the ACK cell in the Table of RSN.

The main objective of these equations is to describe analytically how each message log task may influence the execution time. Equation 4.1 represents the total possible delay that may be introduced in each message transmission. However, it is important to note that the times I_{Log} and U_{Log} spent in the sender (T_Sender) are in its critical path, thus they are forcibly translated into overheads. Considering Equation 4.2, I_{Log} delays the transmission of each message, $Wait\_ACK$ could be overlapped with computations and U_{Log} penalize the progression of the sender process.

Equation 4.3 represents the time that the receiver is blocked. During this time, the receiver can proceed with its execution but it should not send messages to other processes in order to avoid the creation of orphan processes, because messages are just partially logged.

$$Prot\_SBML = T\_Sender + T\_Receiver \tag{4.1}$$

$$T\_Sender = I_{Log} + Wait\_ACK + U_{Log} \tag{4.2}$$

$$T\_Receiver = I_{DataStructs} + Wait\_ACK + U_{DataStructs} \tag{4.3}$$

It is very difficult to represent accurately the recovery cost of a message logging protocol since it depends on the failure moment. In Equation 4.4 we represent the cost of the recovery stage, taking into account that the receiver process has been affected by a failure. *T_Restart* is the time spent in reading the checkpoint from stable storage and restarting the process from it. *T_Broadcast* is the time spent in requesting logged messages to all possible senders. N represents the total number of processes, X represents the number of messages that a process has to send to the restarted process (could be 0 for some processes) and *T_Message* is the time spent in sending a message to the restarted process. *T_Rex* represents the time spent in re-executing the process till reaching the pre-fault state.

$$\begin{gathered} Recovery\_SBML = T\_Restart + T\_Broadcast + \sum_{i=1}^{N}\sum_{j=0}^{X} T\_Message_i + T\_Rex \\ \text{where } i \text{ excludes failed process} \end{gathered} \tag{4.4}$$

The main drawbacks of the SBML are: the complexity in dealing with garbage collection because the message log is distributed among senders; and the disturbances that are generated to senders when processes are being recovered.

When a process is checkpointed, it should communicate to the senders that messages received before the checkpoint will not be needed anymore. This could be a costly operation if there are many senders and a broadcast operation is used. However, some works like [9] propose to piggyback this information when sending an ACK to a sender after a checkpoint.

In faulty scenarios, the failed processes should ask for its logged messages to all possible senders. The senders should stop their execution, or use a separated thread, to serve the logged messages to the failed processes, thus the recovery of failed process affects the execution time of non-failed processes.

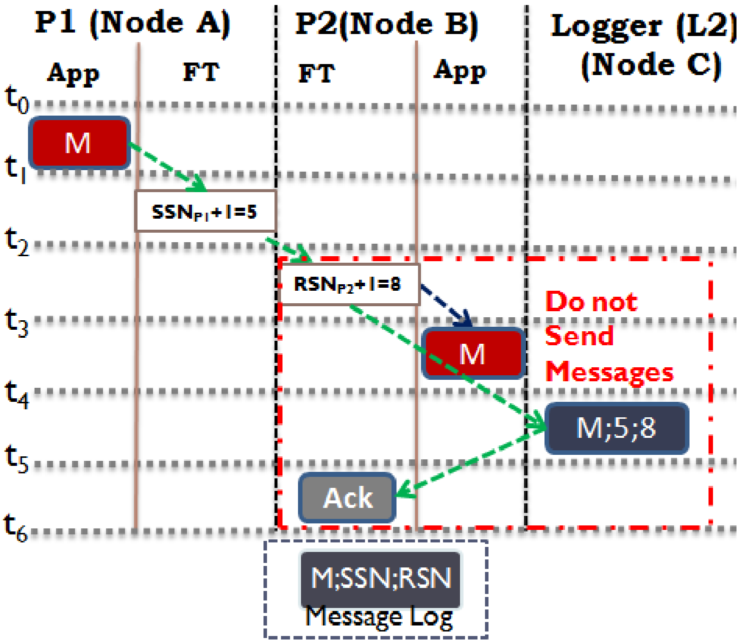


Figure 4.2: Receiver-Based Message Logging.

4.1.2 Receiver-Based Message Logging

In RMBL protocols, is the receiver the one in charge of logging each received message into a stable storage. Thus, in the event of failure, processes are able to reach the same before fault state by reproducing in order the non-deterministic events logged.

The RBML [29] is a solution that introduces more overhead during failure-free executions because each received message should be retransmitted to a stable storage (eg. memory buffer in another node.). This solution may introduce overheads in communications and also could introduce overhead in computations if there are not dedicated resources to deal with message logging.

The main idea behind RBML is to allow failed processes to recover faster by avoiding message requests to non-failed processes and also simplify the garbage collection, since after a process checkpoint all its received messages can be erased from the log which is saved only in one location. In the pessimistic version of this logging protocol, the receiver processes should not send any messages to other processes while all the previous received messages are not properly saved into stable storage. This is done like this in order to avoid the creation of orphan processes in case of failure (Subsection 2.2.3).

In Figure 4.2 we illustrate the RBML operation in its pessimistic version. Here we are considering that there are three processes involved, the sender P1, the receiver P2 and the logger of P2 which is L2. In a system were there are not dedicated resources, application processes will compete for resources with the fault tolerance processes and the logger processes. In Figure 4.2 we are showing in each process and node only the parts involved. We are also assuming that the message M contains all the header information about destination and source.

As can be observed in Figure 4.2, P1 sends a message M with its current SSN, P2 receives M, appends its RSN to it and sends it to a stable storage (L2) (SSN and RSN values are saved together with each checkpoint). Message M is also delivered to the application, it is important to highlight that operations between instants t_4 and t_5 can be overlapped. Then, P2 waits for the confirmation of L2 telling that message M is properly saved in order to allow the application to send new messages. This avoids the impact of partially logged messages in other processes besides the receiver, because if P2 sends and confirms the logging of a message M1 to another process P3 but fails without logging M, then P3 will be an orphan process.

Let us suppose that P2 fails and restarts from a previous state. After restarting, P2 will transfer to its new local node the message log from L2 and consume it in order to reach the pre-fault state. As the non-failed processes that have received a message from P2 have the SSN value of P2, they can easily discard messages that P2 sends during recovery.

The cost of the protection stage in the pessimistic RBML approach is represented in Equation 4.5, where $T\_Forward\_M$ is the time spent in forwarding the received message to a stable storage, such as memory of other node. Then the receiver should wait for the insertion of the message into the message log (I_{Log}) and finally wait for the ACK ($Wait\_ACK$) that indicates that the message if properly saved. During these times, the receiver should not send messages to other processes in order to avoid the creation of orphan processes. In this case we consider that times spent in increasing values of SSN and RSN are negligible, since there is no need to insert values in special data structs.

$$Prot\_RBML = T\_Forward\_M + I_{Log} + Wait\_ACK \tag{4.5}$$

In Equation 4.6 we represent the cost of the recovery stage taking into account that the receiver process has been affected by a failure. $T\_Restart$ is the time spent in copying the checkpoint from stable storage and restarting the process from it. T_Log represents the time spent in transferring the message log saved in stable storage (memory of other node, hard disk, etc.), and T_Rex represents the time spent in re-executing the process till reaching the pre-fault state, the re-execution is usually faster than normal execution since the restarted process has all received messages locally.

$$Recovery\_RBML = T\_Restart + T\_Log + +T\_Rex \tag{4.6}$$

In order to erase old messages, after a process finish its recovery could make a checkpoint and delete all logged messages. This time could also be taken into account to calculate the recovery time.

If we compare Equation 4.1 with Equation 4.5 we can observe that main difference resides in the time that the RBML approach should spent in forwarding each received message. In SBML if the receiver can proceed with computation while the message logging phase is taking place, it would not perceive a high delay.

If we compare Equation 4.4 with Equation 4.6 we can observe that when using RBML, the restarted process will be able to reply its messages independently, relaying only in the messages saved in the log. On the other hand, when using SBML the restarted process will need to consume a message log which is distributed among several senders, and this leads to higher recovery times.

4.2 Hybrid Message Pessimistic Logging

In the previous section, we have briefly described the two main message logging protocols that are used in order to provide FT support to parallel applications. In this section, we present the design and implementation of a Hybrid Message Pessimistic Logging (HM_{PL}) technique which combines the best features of the SBML and the RBML.

The Hybrid Message Pessimistic Logging aims to reduce the overhead introduced by pessimistic receiver-based approaches by allowing applications to continue with its normal execution while messages are being fully logged. In order to guarantee that no message is lost, a pessimistic sender-based logging is used to temporarily save messages while the receiver fully save its received messages.

4.2.1 Key Concepts

The HM_{PL} could be presented as a combination of a pessimistic SBML and an optimistic RBML. We have designed the HM_{PL} to guarantee that no message is lost in presence of failures in order to allow failed processes to reach the same before-failure state. In order to design and develop the HM_{PL} we have set these main objectives:

1. Availability: we focus on providing a strategy that could achieve a Mean Time to Recover (MTTR) of processes similar to the obtained when using a RBML approach. In order to achieve this, we focus on maintaining the fast recovery feature of the RBML by allowing a process to restart and continue with its execution without disturbing non-failed processes.

2. Overhead Reduction: the overheads in communication time during failure-free executions introduced by a RBML technique could be very high [29]. Another source of overhead comes from the blocking behavior of the pessimistic version of SBML and RBML. We have focused on removing these blocking phases from the critical path of a parallel application so we can reduce the overhead introduced.

4.2.2 Design

As we have mentioned, RBML approaches allow fast recovery of failed processes (low MTTR) and SBML approaches introduce low overhead during failure free executions. Taking into account these facts, we have designed the HM_{PL} by combining both strategies. The best way to maintain a low MTTR when using message log is to save messages when receiving them, so the message log is not distributed among several processes. By saving received messages, failed processes may restart from a previous state and then consume this message log without broadcasting requests for past messages to non-failed processes. The problem with receiver-based approaches is that each received message should be forwarded to stable storage (eg. memory of other node) and this increases the message transmission and management times.

On the other hand, SBML approaches introduce less overhead during failure free executions since messages may be saved in a buffer of the sender. If a process fails and restarts, it has to ask for all necessary messages to the senders in order to reach the pre-fault state, thus increasing the recovery time. Garbage collection is more complex in SBML since a checkpointed process should notify the senders so they can erase old messages that belong to the checkpointed process.

Figure 4.3 shows the basic operation of the HM_{PL} when messages are sent from one process to another (P1 to P2). We have introduced a new data structure which is a temporary buffer. This buffer is used in the sender and also in the receiver to temporally save messages in order to allow the receiver to communicate without waiting for the message log protocol to finish the full cycle. Thus, the HM_{PL} removes the blocking behavior of SBML and RBML while the logging is taking place, this means that receiver processes can communicate with others while messages are just partially logged. The logger

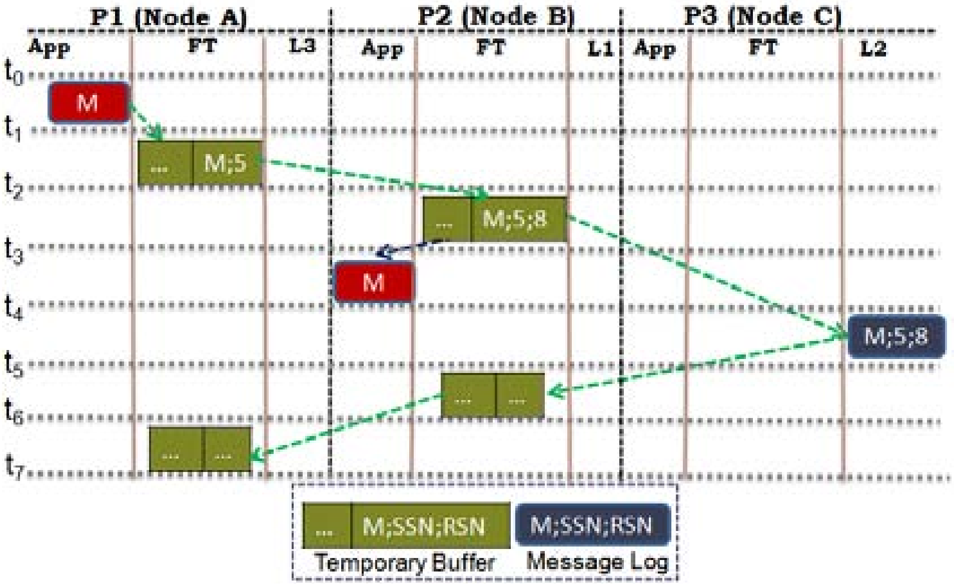


Figure 4.3: Hybrid Message Pessimistic Logging

L2 stores the messages in an array in memory which is flushed to disk asynchronously when a memory limit is exceeded or opportunistically. We consider that message M has all the information about sender and receiver.

Figure 4.4 shows the flow diagram of the HM_{PL} during the protection stage. It is important to highlight that we assume the utilization of a transparent fault tolerant middleware that intercepts and manages messages (FT column in the figure). Before sending a message, FT of P1 inserts the message M with its SSN in its temporary buffer (TB). When FT of P2 receives M (checks the SSN to discard already received messages) it inserts M, the SSN and RSN in the TB and proceed with the normal execution. In the meantime, the FT process of P2 will transmit the message and all the extra data to L2 which is located in another node, and once FT of P2 receives the confirmation that M has been correctly saved, it will erase M from its TB (t6 in Figure 4.4) and inform FT of P1 so it can erase M also from its TB (t7).

Figure 4.5 illustrates the flow diagram of the recovery procedure of the receiver. FT of P2 will receive the message log and checkpoint and will restart the application, connect to its new logger and consume the message log saved by the Logger L2. After finishing with this, FT of P2 will ask P1's FT if it has a message in its TB and consume it. Then, the normal execution will continue, but when P2 has to receive the first message after recovery from a sender, P2's FT will ask if there are not messages in sender's TB.

It is important to highlight that in most cases the receiver will be able to fully recover using its message log, however when there are partially logged messages it will need to ask

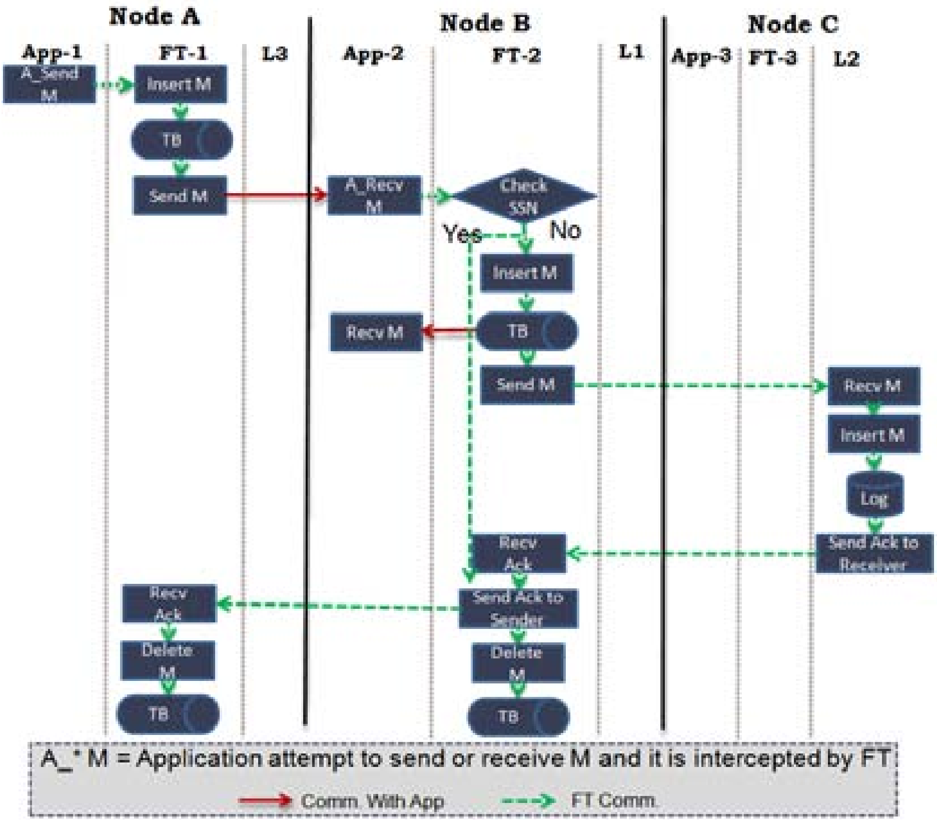


Figure 4.4: Hybrid Message Pessimistic Logging. Protection mechanism.

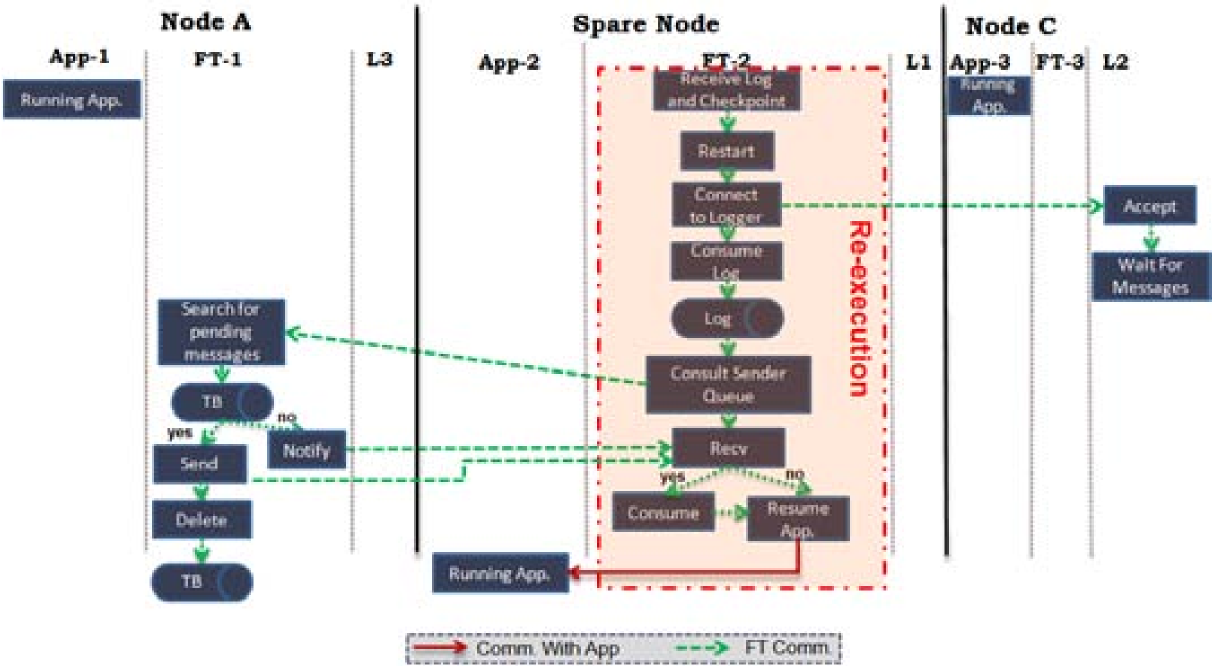


Figure 4.5: Hybrid Message Pessimistic Logging. Recovery of the receiver.

for messages in the TBs of senders. The senders will almost not be affected by requests since the receiver may access directly the TB of the senders through the fault tolerance middleware and copy messages.

Taking into account the sender side of each process, when a process is restarted from a previous checkpoint it may re-send some messages that the FT process will transparently discard according to the current SSN that the receiver has for that sender. Furthermore, the receiver can inform the sender of the current value of the SSN, so the sender will avoid resending unnecessary messages.

Let us analyze the failure moments and what is the impact on each situation (we focus on the failure of the receiver). In order to explain the failure moments, we will use the logical times (t_x)of Figure 4.3:

- *t2-t4*: P1 s FT has the message in its TB, so once P2 starts re-executing it will consume all saved log in L2, and then ask P1 for message M.
- *t4-t6*: P1 s FT has the message in its TB, also L2 has saved M but do not send the ACK to P2, so L2 will erase this message and P1 will re-send M to P2 during re-execution.
- *From t6*: M has been saved in L2 and confirmed to P2 s FT. P2 will consume this message from the message log. In situations where M has not been erased from P1 s queue, after restarting P2 will continue with the logging procedure where it stopped and will send the ACK to P1 so it will remove M from its TB.

The overhead of the protection stage in the HM_{PL} is modeled in Equation 4.7, where I_{TB} is the cost of inserting each message in the TB of the sender or the receiver. Equation 4.8 shows the times that are not in the critical path of each message transmission (unlike the RBML) when using HM_{PL}, where $T\_Forward\_M$ is the time spent when retransmitting the message M to the Logger in another node, I_{Log} is the time spent in introducing the message in Log and $Wait\_ACK$ is the time spent in waiting ACKs from other nodes. We are not taking into account the times spent in increasing SSN and RSN values or checking them, since these times could be negligible.

$$Prot\_HM_L = 2 * I_{TB} + T\_Overlapped \tag{4.7}$$

$$T_{Overlapped} = T\_Message + I_{Log} + 2 * Wait\_ACK \tag{4.8}$$

In Equation 4.9 we represent the cost of the recovery stage, where $T\_Restart$ is the time spent in copying the checkpoint from another node and restarting the process from it, T_Log represents the time spent in copying the message log from stable storage (e.g. memory another node). M represents the number of neighbors (senders) that have a message for the restarted process in their TB. $T\_Message$ is the time spent in copying messages from senders. T_Rex represents the re-execution time and $T\_ConnectLogger$ is the handshake time between the restarted process and the Logger.

$$Recovery\_HM_L = T\_Restart + T_{Log} + \sum_{i=1}^{M} T\_Message_i + T\_Rex + T\_ConnectLogger \quad (4.9)$$

The main objective of these equations is to describe analytically the operation of the HM_{PL}.

4.2.3 Orphan Processes

As the HM_{PL} combines a pessimistic SBML with an optimistic RBML there may be a possibility of creating orphan processes. Here we explain how we avoid the creation of orphan processes. In message logging, the order of reception is considered the unique source of non-determinism. When using MPI to write parallel applications, the main source of non-determinism is the usage of the wild card MPI_ANY_SOURCE to receive a message. This tag allows the reception of messages from any possible sender, without specifying a particular one.

Let us consider the situation in Figure 4.6, where message M1 is sent from P1, received in P2 (R1) and logged in L2 (L_M1). Assuming that reception R2 in P2 is done with a wild card, then P2 receives M2 from P1. Suppose that having received the message M2, P2 sends M3 to P4 (R3) and then fails without logging M2 in its logger.

P2 restarts from its checkpoint ($C_{2,1}$) and reads M1 from its log (RL1) and then instead of consuming M2 from the temporary buffer of P1, consumes M2 from the temporary buffer of P3. Then, P2 saves M2 in its logger with operation L_M2 and produces M3 instead of M3 making P4 an orphan process.

In order to avoid the explained situation, when we detect that a wild card reception is being used, we do not allow the receiver to continue till the message is fully logged. This also allows us to avoid the generation of requests to all processes temporary buffers when restarting.

As different senders have probably initiate the communication to a wild card reception

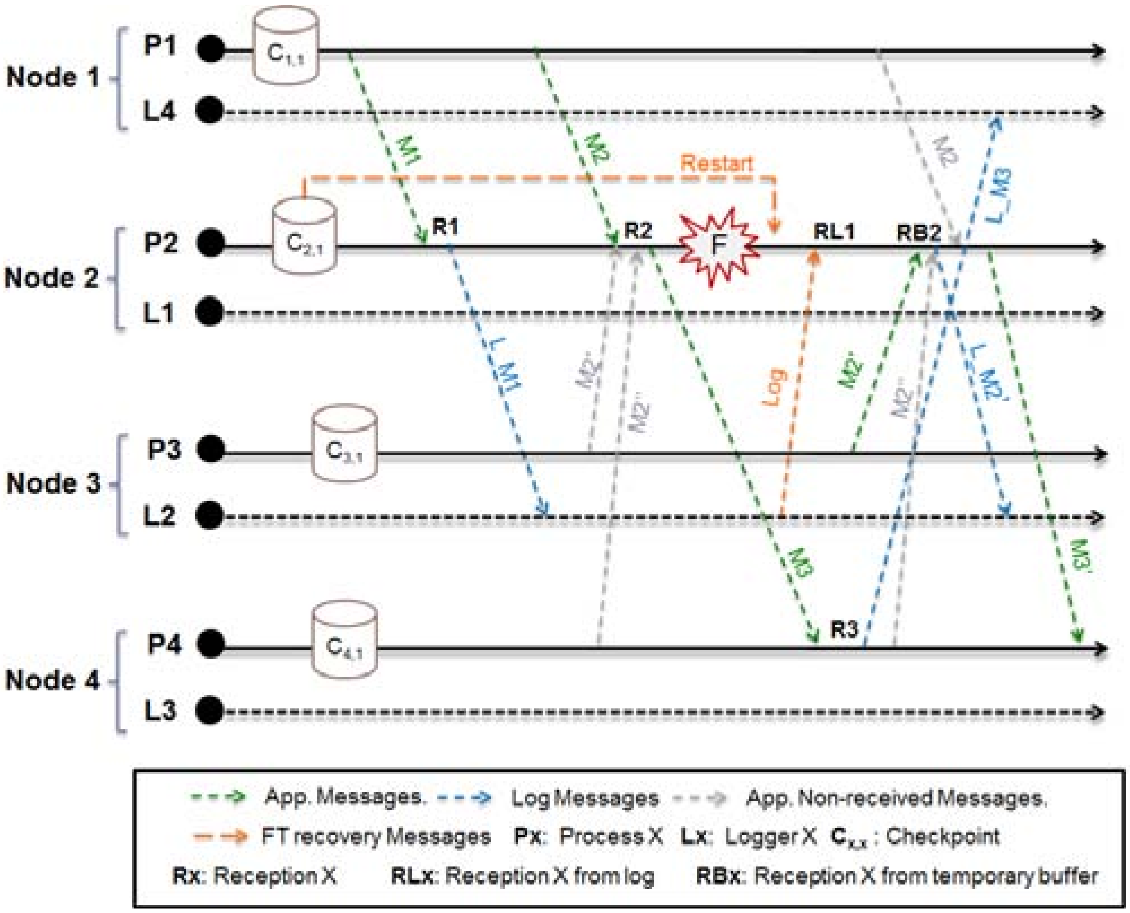


Figure 4.6: Orphan Processes in the Hybrid Message Pessimistic Logging.

(M2, M2' and M2"), non-confirmed messages by the receiver will be erased from the TB of the senders after the receiver process confirms the reception of a subsequent message.

4.2.4 Implementation Details

We have included the HM_{PL} inside the RADIC-OMPI implementation (Section 2.3.4). The HM_{PL} has been included in the *Vprotocol* Framework of Open MPI since this framework enables the implementation of new message logging protocols in the Open MPI library [11]. The main components of our message logging implementation are described below, as the HM_{PL} is a combination of sender and receiver approaches we split the functionality to explain it:

1. *Sender Message Logging*: Before sending a message the the payload of the message is saved in a circular queue (temporary buffer) in memory. As this circular queue will be continuously modified, there is no need to flush it to disk. Before sending another message to the same receiver, the non-finished logging transmissions will

be checked in order to erase messages from the circular queue. Normally, the circular queue will contain at most one message per receiver (neighbors). However, in order to guarantee that the size does not grow uncontrollably, the circular queue size is limited according to a percentage of the total memory available in the node.

2. *Receiver Message Logging*: When a message is received, it is introduced inside a circular queue and then the message is sent to the logger residing in another node by using a non-blocking communication. When the logger informs that this message has been saved it will remove it from its circular queue. Before receiving a message the SSN is always checked in order to discard already received messages.

3. *Logger*: We have added special threads (one per application process) that are executed outside the communicator of the parallel application. Each logger publishes its name so an application process can get connected to a logger residing in a different node when finishing the MPI_Init command . When a message is received from a connected process, the logger will save this message in its volatile memory until a defined level of memory is consumed, then it will asynchronously start to flush data to disk or it can also command a checkpoint because a memory limit was exceeded.

4.3 Experimental Validation

To carry out the implementation and experimental validation of the proposed technique, we selected the Open MPI library as an implementation of MPI standard, and we implemented RADIC (described in section 2.3) and different message log techniques inside the library. The RADIC middleware is the one in charge of all message transactions that are made through the MPI library. Then, all messages are intercepted and treated according to the specification of each described message logging technique. Details about the experimental environment can be found in section 6.1.

In order to make the first experimental validation of our HM_{PL}, we have used the NetPipe tool [81]. We have compared the HM_{PL} with a classic pessimistic receiver-based message logging technique. We have executed the netpipe tool with using two processes in different machines, where one of the process acts as a sender and the other as a receiver.

When using the receiver-based logging, every time a message is received, it is first forwarded to a logger thread residing in another node, and only after the message is fully saved, the MPI receive call finalizes. On the other hand, as the HM_{PL} proposes the utilization of temporary buffers, there is no need to wait for messages to be saved while they are being transmitted to logger threads, thus after receiving each message it is copied to a temporary buffer and then the MPI receive call can finalize.

Figure 4.7 and Figure 4.8 show the bandwidth utilization and the overheads respectively, when using each of the message logging techniques described. We should notice here that the benchmark executed without message logging achieves the 100% of bandwidth utilization. We are showing results between 64 KB and 64 MB message sizes. As the default eager limit of the MPI library is set to 64 KB, for messages smaller that this size there is no difference between both message logging techniques. The eager limit is the max payload size that is sent with MPI match information to the receiver as the first part of the transfer. If the entire message fits in the eager limit, no further transfers / no CTS is needed. If the message is longer than the eager limit, the receive will eventually send back a CTS and the sender will proceed to transfer the rest of the message.

In Figure 4.7 can be observed that the bandwidth utilization with the HM_{PL} remains near to 85% while with the receiver-based logging only reachs 50%. Figure 4.8 shows that the overheads with the HM_{PL} are near to 20% for each message transmission and with the receiver-based logging is near to 100%, since for each sent message the application should wait the double time for an answer.

It is important to highlight that these results are just taking into account two independent processes running in two different nodes, and one logger thread in other node. Moreover, the system is not under full utilization and the network cards are not overloaded. Also, the NetPipe tool just gives us an approximation of the logging overheads taking into account the communication times but not the computation cost of logger threads. Thus, the added overhead will depend also on applications behavior and usage level of the system.

Experiments with more complex and full featured benchmarks are presented in section 6.3.

4.4 Discussion

Throughout this chapter, we have described the proposed Hybrid Message Pessimistic Logging (HM_{PL}). This is a novel message logging approach which combines the advantages of two of the most classical message logging approaches: Sender-based Message Logging and Receiver-based Message Logging. The HM_{PL} focuses on providing a fault tolerant solution with low MTTR by accelerating the recovery process of Sender-based approaches

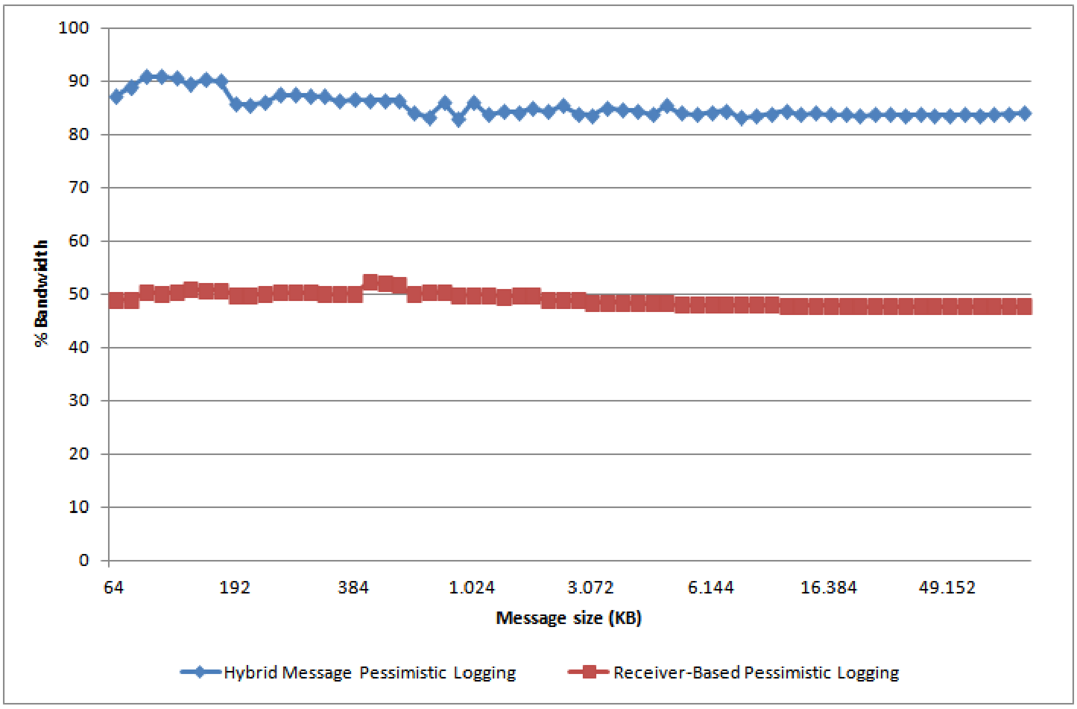


Figure 4.7: Bandwidth utilization obtained with NetPipe Tool.

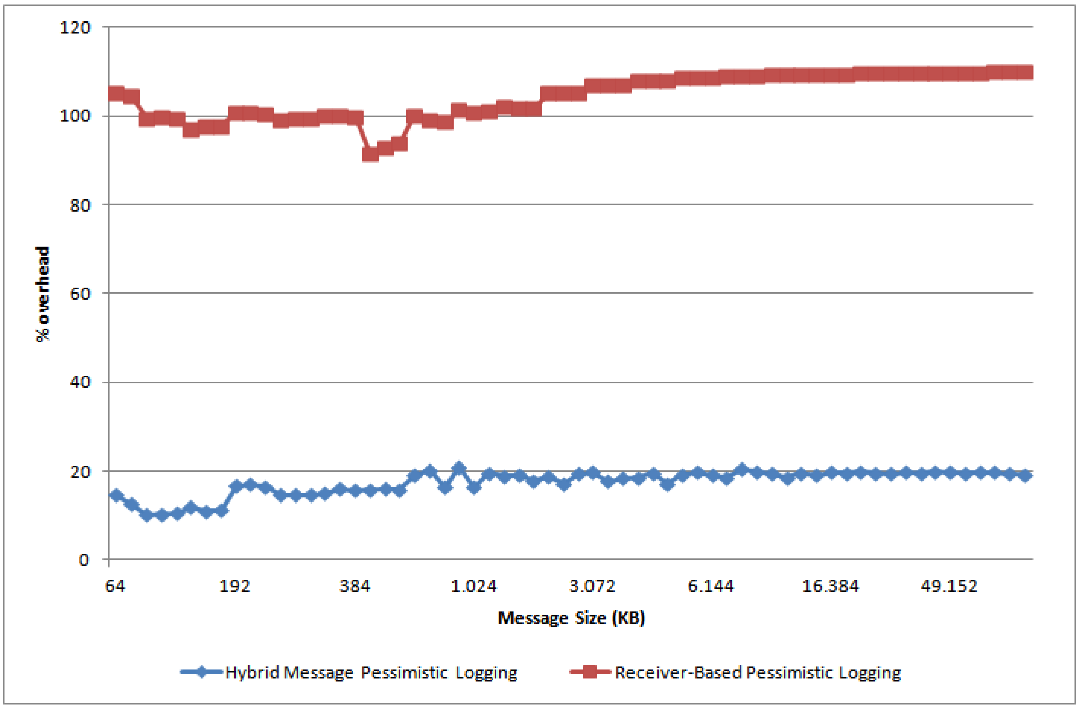


Figure 4.8: Overheads in message transmissions obtained with NetPipe Tool.

and at the same time reducing the impact of failure-free executions in comparison with receiver-based approaches.

This work relies on the usage of data structures to save messages temporarily (in senders and receivers) and allowing the application to continue its execution without restricting message emissions while other messages are being saved in stable storage.

As has been described, sender-based message logging is the solution that introduces less overhead during failure-free executions since messages are saved in the volatile memory of senders avoiding the retransmissions of messages. However, sender-based logging is considered as one of the more costly protocols during the recovery phase, since sender processes should retransmit old messages in order. Also garbage collection is complex since after each checkpoint, processes should notify their senders to erase old messages.

In receiver-based message logging, failure-free executions are slower since each received message should be retransmitted to stable storage. However, this protocol provides faster recovery since the message log is available only in one location and there is no need for senders collaboration. Garbage collection is easier in these protocols since after a checkpoint a process can delete its logged messages.

To the best of our knowledge, there are not message logging protocols that combine low failure-free overhead, fast recovery and low complexity in garbage collection.

Chapter 5

Balancing Dependability and Performance in Parallel Applications

In this chapter, we will analyze the impact of fault tolerance techniques in parallel applications. As we have presented in previous chapters, many efforts have been made in order to design new fault tolerance techniques to lower the overheads in parallel executions. Another important matter that should receive attention is application configurations. Therefore, resource consumption and overhead management should be taken into account.

In subsection 2.3.3, we have described the inclusion of spare nodes in RADIC, and how this benefits performance. We believe that the first step to guarantee the maintenance of computation capacity after failures is by replacing failed nodes instead of overloading nodes. By doing this process automatically, we can reduce the MTTR.

The usage of rollback-recovery based fault tolerance techniques in applications executed on multicore clusters is still a challenge, because the overheads added depend on the applications behavior and resource utilization. Many fault tolerance mechanisms have been developed in recent years, but analysis is lacking concerning how parallel applications are affected when applying such mechanisms.

The first contribution explained in this section addresses the combination of process mapping and fault tolerance task mapping in multicore environments. The main goal of this contribution is to propose a methodology to analyze the mapping of fault tolerance tasks and help to determine which one generates the least disturbance to the parallel application by characterizing the application behavior and the impact of fault tolerance tasks.

As a second contribution in this section, is presented a methodology that allows to obtain the maximum speedup under a defined efficiency threshold taking into account the impact of a fault tolerance strategy when executing parallel applications in multicore clusters. This is a study case and this methodology is designed for Single Program Multiple Data (SPMD) applications.

5.1 Parallel Applications in Multicore Environments

Current parallel applications that are executed in High Performance Computing (HPC) clusters try to take advantage of the parallelism in order to execute more work in a smaller amount of time. Main objectives when running applications on parallel environments are: maintaining speedup as close as possible to the ideal, maintaining scalability with an efficient utilization of the available resources allowing applications to finish successfully. In order to provide resiliency to parallel applications, rollback-recovery based fault tolerance seems to be the best option. However, the usage of these techniques in multicore clusters is still challenging, because the overheads depend on applications behavior and resource utilization.

HPC clusters are now composed of multicore machines because this gives greater computing capacity [69]. However, they have heterogeneous communication levels to perform the MPI communication (cache memory, main memory and local area network). These paths present different latencies and bandwidths, which have to be considered when parallel applications with a high communication rate and synchronism, such as SPMD, are executed in these clusters.

In Figure 5.1 we present three main possible scenarios when mapping applications in multicore systems. It is important to highlight that in this figure we are considering one iteration of a SPMD application. In Figure 5.1, we decompose the Single Stream in communication and computation operations. We are assuming that the send operation has almost no effect in computations, and that the receive operation takes place while the computation continues. The main scenarios are:

1. **Communication bound**: Applications in which the processes are waiting because the communications take more time than computations belong to this scenario. Figure 5.1a shows how a communication bound application behaves. In this figure, we focus on showing how reception times (non-blocking send operations do not delay considerably the execution) can highly influence the execution time of a parallel application.

2. **Balanced Application**: This scenario is the best regarding efficient resource utilization, because the computational elements are working while the communication takes place. However, this behavior is very difficult to obtain because

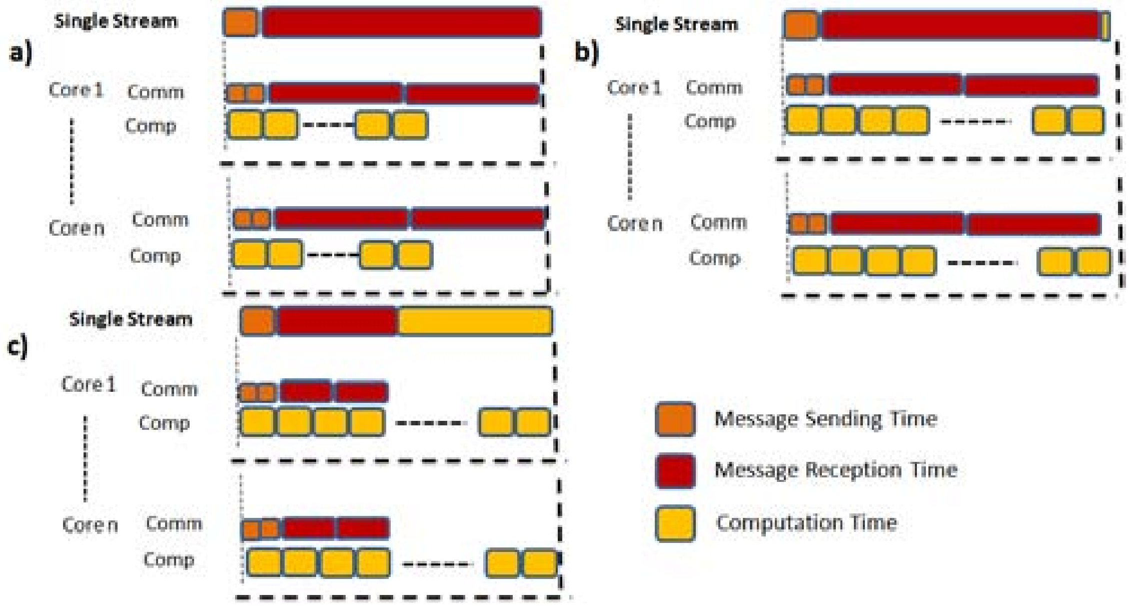


Figure 5.1: Parallel Executions Scenarios in a SPMD App. a)Communication Bound. b) Computation and Communication overlapped. c) Computation Bound.

a previous study of the application is needed in order to completely overlap computations and communications (Figure 5.1b).

3. **Computation Bound**: When operators try to make a good use of the parallel environment, they try to maintain the CPU efficiency high. Then in order to avoid the communication bound scenario it is recommended to give a bigger input per process which usually leads to a computation bound scenario. Figure 5.1c illustrates this scenario.

When characterizing a parallel application, it is also important to take into account the number of processes that will be used, the number of nodes and the memory consumption of each process. This analysis should be done in combination with the analysis of the parallel environment in order to determine resource utilization. When fault tolerance support is going to be used, it is important to consider it as part of the environment.

Mapping an application and tasks from a fault tolerant middleware into a multicore platform requires communication, concurrences and synchronization controls of all components involved. Figure 5.2 illustrates the possible effects of message logging during application executions. If message logging tasks are not considered when configuring a parallel execution, the application could make an inefficient use of resources. Then, when considering message logging tasks impact we can decide to choose between sharing cores with the application processes, or save cores for them. The election of the fault tolerant

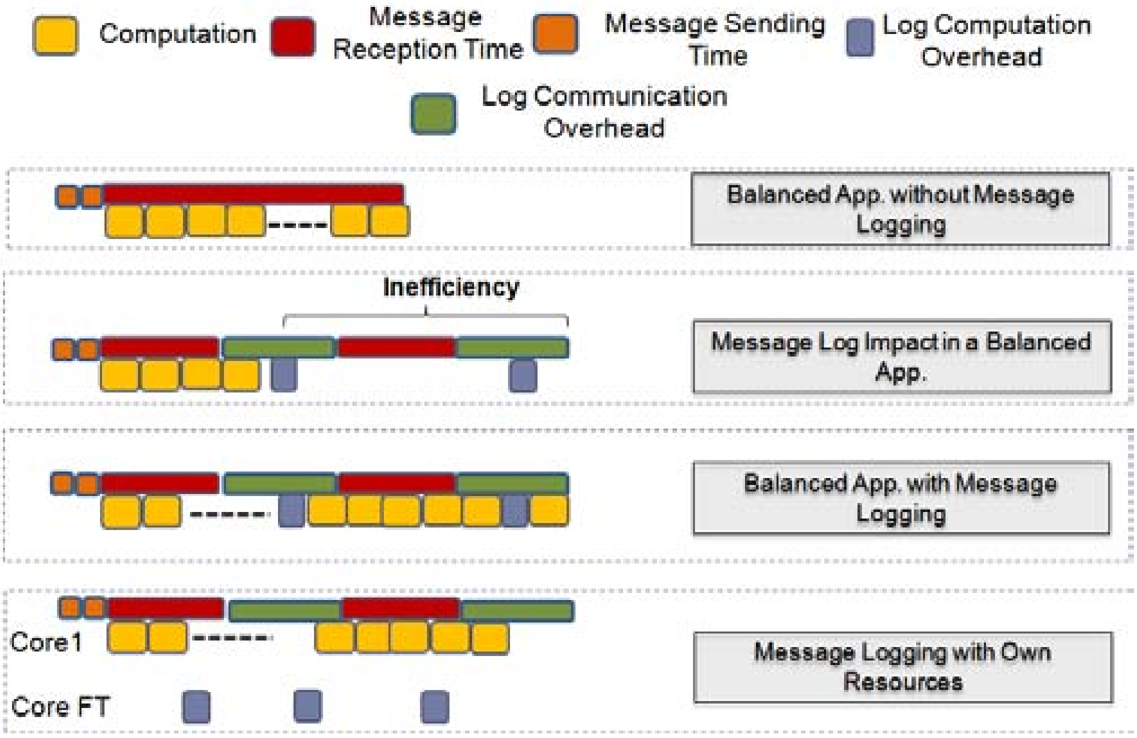

Computation
Message Reception Time
Message Sending Time
Log Computation Overhead
Log Communication Overhead
Balanced App. without Message Logging
Inefficiency
Message Log Impact in a Balanced App.
Balanced App. with Message Logging
Core1
Core FT
Message Logging with Own Resources

Figure 5.2: Message Logging Impact Analysis in Parallel Applications.

configuration and mapping must be made taking into account the application and the specific platform with its advantages and limitations.

5.2 Message Logging Processes Mapping

In this section, we address the combination of process mapping and fault tolerance task mapping in multicore environments. The main goal is to determine the configuration of a message logging approach which generates less disturbance to the parallel application. We propose to characterize the parallel application in combination with the message logging approach. This allows us to determine the most significant aspects of the application such as computation-communication ratio and then, according to the values obtained, we suggest a configuration that can minimize the added overhead for each specific scenario.

In order to find the most appropriate configuration of the message logging approach, we should analyze how the parallel application and the logging approach coexist in the parallel machine. There will be two parallel applications that will compete for resources, thus it is critical to analyze the influence of fault tolerance in application behavior.

We propose to analyze parallel applications and obtain information that allows us to configure properly the fault tolerance tasks, specifically we determine if the best option is to share (compete for) resources with application processes or save resources for the fault tolerance tasks in order to reduce the introduced disturbance.

5.2.1 Message Logging Tasks

Here we are going to describe and analyze the main tasks of message logging techniques. For the analysis that we do in this section we consider a pessimistic receiver-based message logging protocol, however other message logging approaches could use our methodology. We will focus only in message logging tasks because they are responsible for most of the overhead in failure-free executions when using uncoordinated checkpoint approaches.

Most of the impact of a pessimistic receiver-based message logging protocol concentrates on communications and storage (memory or hard disks), but there is also an impact on computations because fault tolerance tasks also need some CPU cycles in order to carry on their work.

Considering the RADIC architecture, we will focus on the fault tolerance tasks carried on by the Protector component. Protectors main functions during the protection stage are: establish a heartbeat/watchdog mechanism between nodes (low CPU consumption operation, do not depend on application behavior), to receive and manage message logs (CPU consumption depends on application) and checkpoints from observers (infrequent operation).

All communications between processes when using RADIC go through Observers and each received message is sent to a corresponding logger thread. In Figure 5.3a we show the default layout of the RADIC architecture, where all the protectors functionalities are carried on by a single process that steal CPU cycles randomly from each available core.

However, some other options may fit better to a multicore environment. Let us divide the protector s tasks in two: the heartbeat watchdog procedure (1 thread) and the message logging procedure (1 thread for each protected process). Figure 5.3b shows an homogeneously distribution of logger threads among cores and Figure 5.3c shows how we can save resources for the protector tasks to avoid context switches between logger threads and application processes.

5.2.2 Analyzing Parallel Applications

Regarding the configuration and mapping of parallel application tasks and message logging tasks into a multicore node, we consider that three different strategies can be used:

- OS-based: in this case, the scheduler of the operating systems takes care of the distribution of tasks among the available cores. This is a default strategy, but

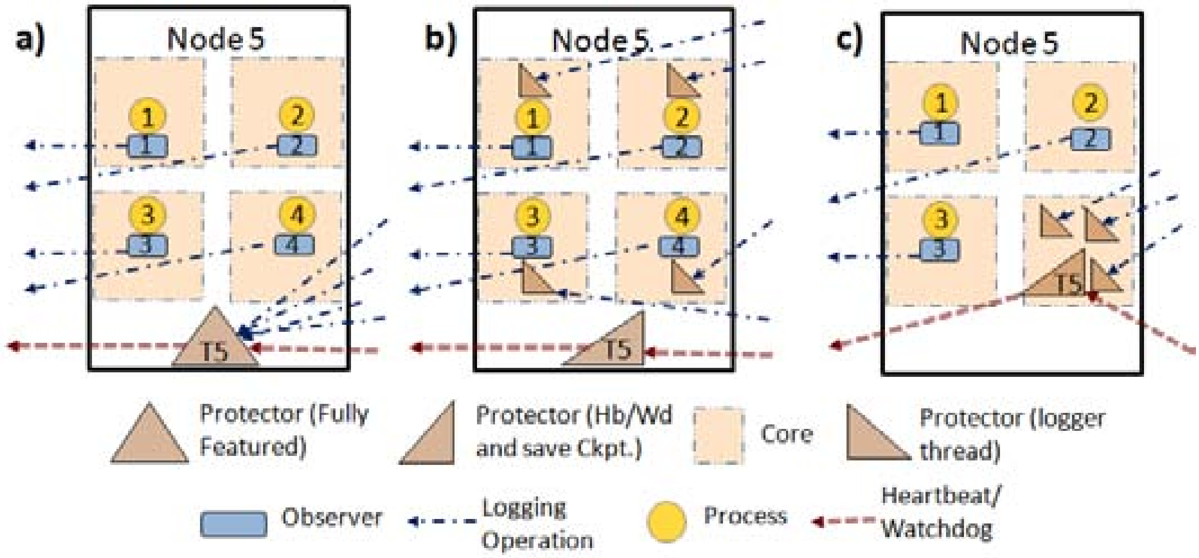


Figure 5.3: Parallel application with RADIC fault tolerance processes. a)RADIC default configuration. b) Logger threads distributed among available cores c) Protectors processes with own computational resources.

when there might be competition between tasks, imbalance problems may appear.

- Homogenous distribution: in this case, we can manually distribute the tasks among the available cores searching for a homogenous load distribution. This strategy focuses on guaranteeing that all cores execute almost the same amount of work.
- Own FT resources: this strategy consists on saving some cores in order to map the message logging tasks there, avoiding the context switch between them and applications processes.

In order to reduce the impact of the pessimistic receiver-based logging protocol we propose to analyze which of the three different strategies fits better to a particular application behavior. It is important to consider that according to this message logging protocol, every message should be logged in a different computing node. Therefore, there might be a considerable increase in the transmission time of each message when the message logging protocol is activated. Thus, when executing a parallel application with message logging the processes will be waiting a longer time for each message and the computation will be affected by the logger threads.

In order to reduce the overheads of the message logging approach, we analyze how the application will be affected when introducing this logging technique. In Figure 5.4 we can observe two main scenarios that could appear when we have a single protector program

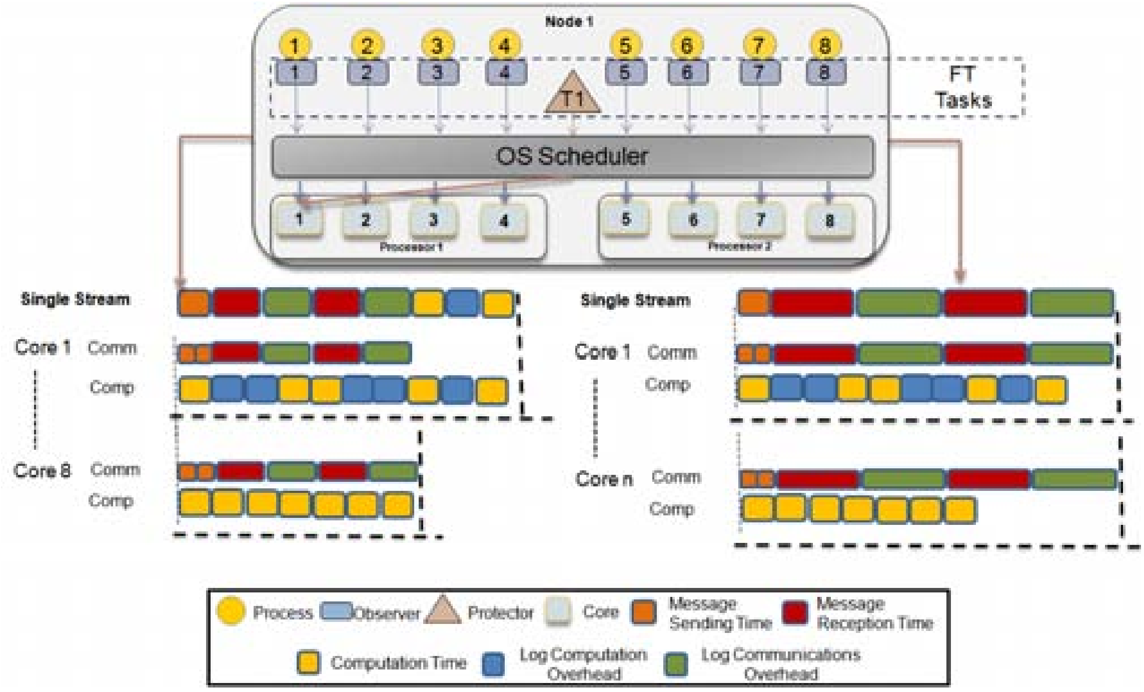


Figure 5.4: Automatic Distribution of Fault Tolerance Tasks.

(single logger) that receives messages from different processes. Depending on the application behavior, we can have a computation bound scenario (left of the figure), a communication bound scenario (rigth side of the figure), or in the best case a balanced scenario. In this case, we consider that the OS scheduler is assigning the tasks as convenient, thus the iteration time of processes may differ or we can have a low efficiency value. In these two scenarios, the message logging overheads cannot be hidden transparently to the application, but when a distributed logger model is used in computation bound applications, we can manage the mapping of the logger threads to distribute the overheads among processes.

Figure 5.5 shows a second option of message logging tasks mapping. In this case, we opt to equally distributed the overhead of the logger threads among the available cores (we consider that the HB/WD thread overhead is negligible so we let it to the OS scheduler). In a computation bound scenario, an homogenous distribution of overhead may cause that the computation time increases enlarging the iteration time.

Alternatively, as can be observed in Figure 5.6 we can choose to save some computational cores in each node in order to avoid context switch between application processes and logger threads. In this case, we can have smaller iteration time by avoiding the disturbance in application processes.

Then, to determine a suitable configuration for message logging tasks is necessary to extract the communication-computation relationship, the communication pattern and also

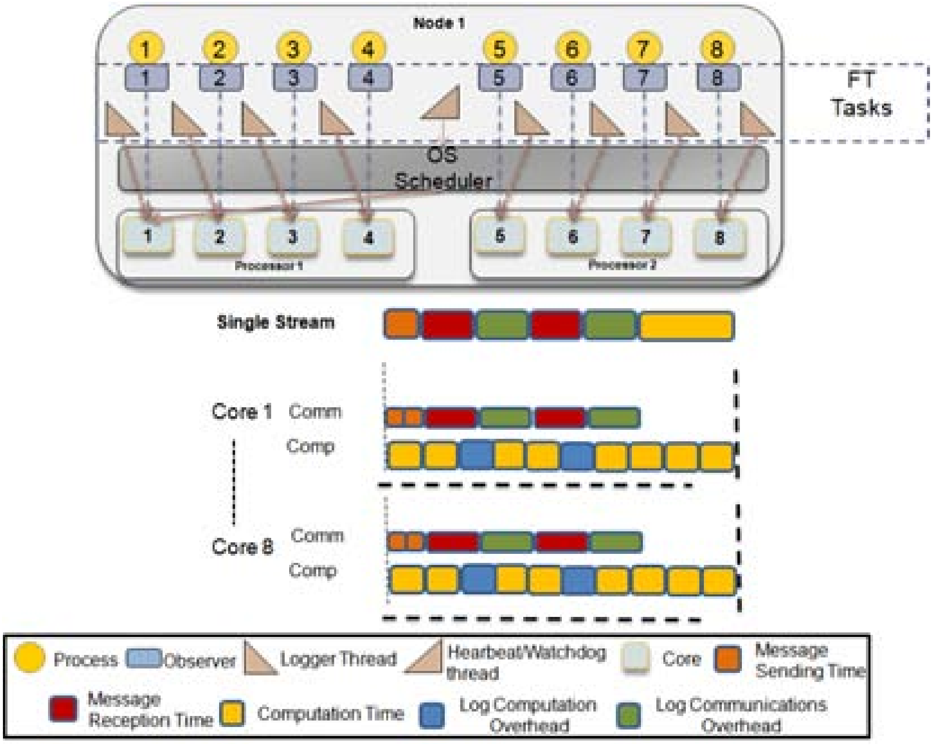


Figure 5.5: Homogenous Distribution of Fault Tolerance Tasks.

the original mapping of the application. It is important to obtain the mapping so we can now if there are available resources inside nodes to use them for the message logging tasks.

5.2.3 Methodology to Determine Suitable Configurations

Considering that many parallel applications are executed with balanced per-core workload, our default proposal is to distribute the overhead in computation produced by the logger threads among application processes residing in a node, as has been observed in Figure 5.5.

However, as not all applications are configured with a balanced per-core workload we propose a methodology to determine the most suitable configuration of message logging tasks, this methodology is shown in Figure 5.7. We propose to characterize the application in order to extract the communication-computation ratio and the communication pattern. To carry out these characterization phase, we propose to use a few iterations of the application or to extract application traces.

In order to extract application traces and to determine transparently the communication pattern we propose to use the PAS2P (Parallel Application Signature for Performance Prediction) tool [85]. This tool extracts the main phases of a parallel application by collecting and analyzing computation and communication events. This tool also allows us

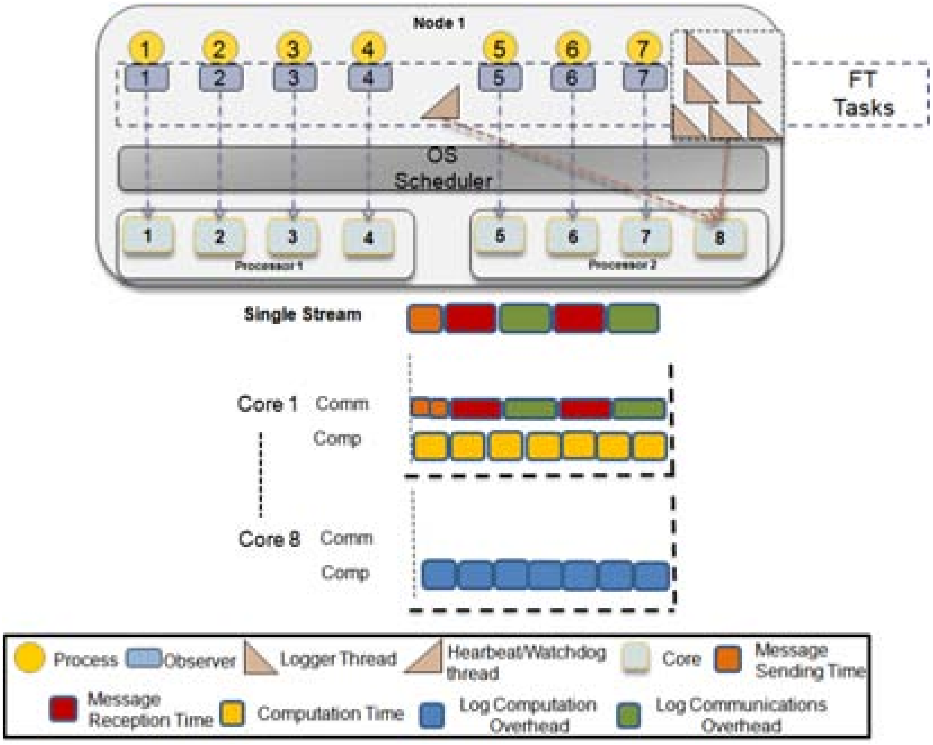


Figure 5.6: Saving Cores for Fault Tolerance Tasks.

to extract a signature that represents the application execution and we use this signature in order to analyze the impact of the message logging tasks in the application by executing this signature. The execution of the signature may represent around 5% of the total execution time, depending on the application length.

Once the parallel application behavior is extracted, we can execute the signature or application kernel in combination with the loggers threads, and analyze each configuration described previously (OS-based, Homogenous Distribution, Own FT resources). If we determine that the default proposal of sharing resources between message logging tasks and application processes make the application behaves as showed in Figure 5.5, we propose saving cores in each node in order to assign them to the logger threads, obtaining the behavior showed in Figure 5.6.

As saving cores may make the initial mapping change if there were not available cores for the logging tasks, we should also analyze if the new mapping does not affect negatively the execution, resulting in a worse performance than the default option. If we can modify the application parameters, in order to analyze the new mapping, we use a kernel of the parallel application and execute a set of iterations to discover any increase in the execution time. However, when we can only obtain traces of parallel applications, we analyze the mapping changes with the PAS2P tool.

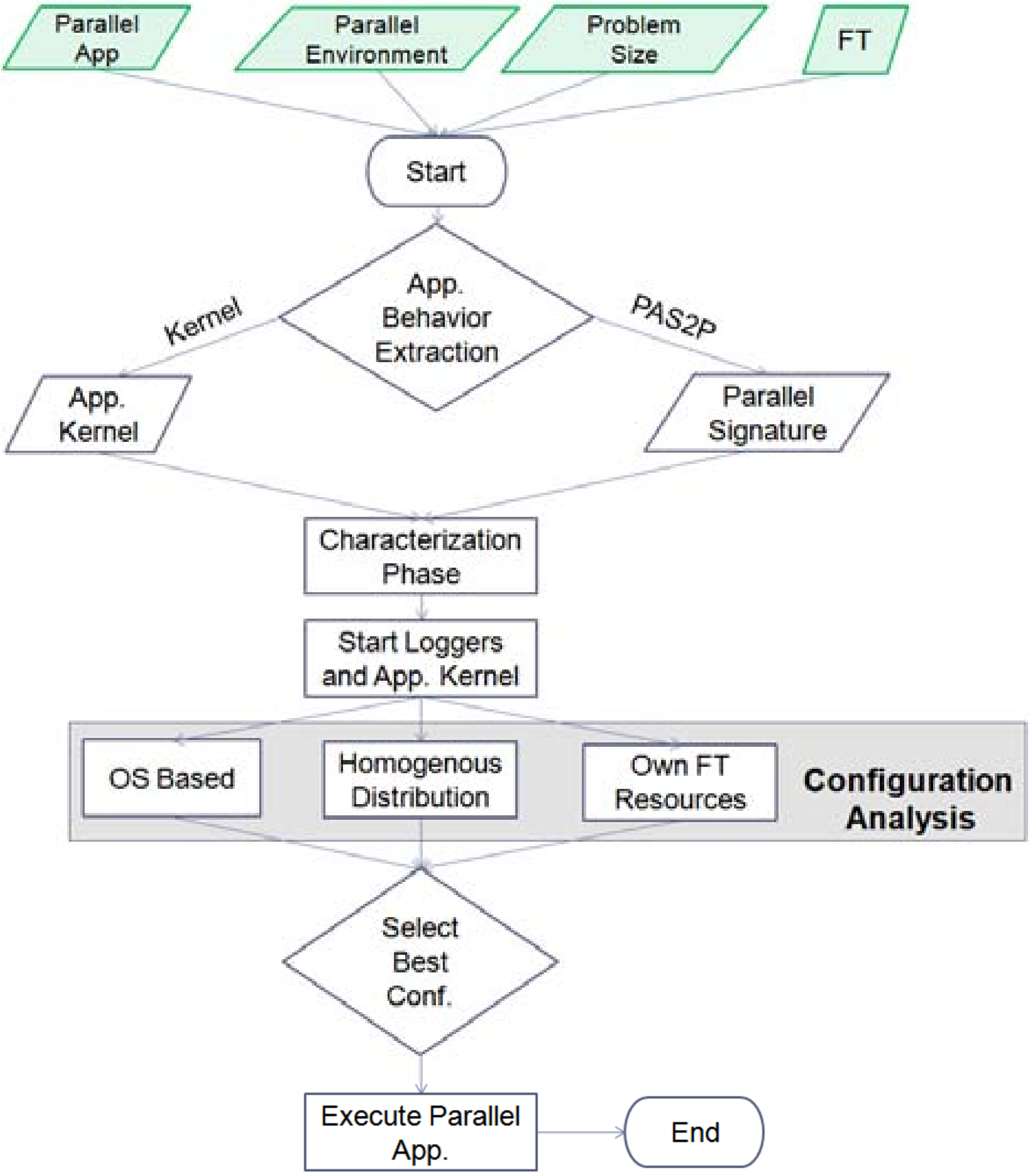


Figure 5.7: Flowchart of the Methodology to Determine Suitable Fault Tolerance Tasks Configuration.

Another important aspect that we characterize is the per-process memory consumption. This is significant because it may be a good choice to save logged messages in memory instead of hard disks, as this allows us to avoid bigger delays when storing messages. When we put less processes per node, we can save more memory for the message log, thus there is more time to overlap the flush-to-disk operation with receptions of new messages to log. Also, we can use longer checkpoint intervals if we consider an event-triggered checkpoint mechanism where a full message log buffer triggers a checkpoint.

When we choose to save cores for the logging mechanisms, we propose to save 10% of the available cores in the node. Initial analysis have shown that the CPU consumption of the logger threads are near 10%. However, we are aware that this depends on the message sizes and frequency. As a future work we will focus on proposing a methodology to determine CPU consumption of the logging operation in order to save a suitable number of cores per node.

5.2.4 Methodology Validation

Here we validate the methodology presented previously. As testbed we use a Heat Transfer application and a Laplace Solver application. Both follows a SPMD communication pattern and both applications allow overlapping between the computation steps and the communication steps and both are written using non-blocking communications. Details about the experimental environment can be found in section 6.1.

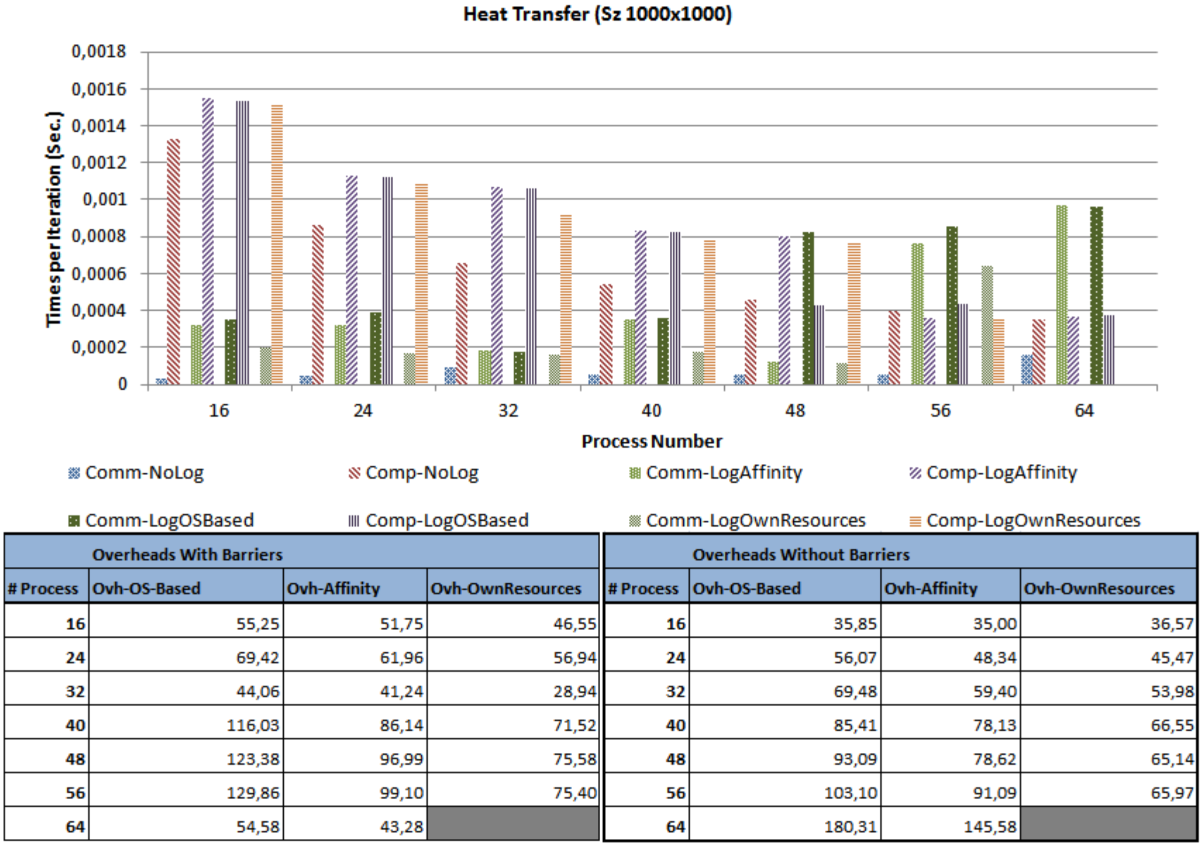
The computation and communication times per iteration showed in bars in Figure 5.8 are obtained by executing a few iterations of the parallel applications observing all processes and then selecting the slowest one for each number of processes. The total execution times have been obtained using 10000 iterations.

In these experiments we have only considered the overlapped times (communication and computations) because they represent the higher portion of both applications. We have discarded the delays caused by desynchronization and the computation time spent in computing the edges of each sub-matrix before sending it to the neighbors.

For both applications we have measured communication and computation times with the following options:

1. Without using message logging (Comm-NoLog and Comp-NoLog).

2. With message logging using all available cores in each node and giving affinity to each logger thread in order to ensure an equally distributed overhead in computation among all application processes (Comm-LogAffinity and Comp-LogAffinity).



Overheads With Barriers				Overheads Without Barriers			
# Process	Ovh-OS-Based	Ovh-Affinity	Ovh-OwnResources	# Process	Ovh-OS-Based	Ovh-Affinity	Ovh-OwnResources
16	55,25	51,75	46,55	16	35,85	35,00	36,57
24	69,42	61,96	56,94	24	56,07	48,34	45,47
32	44,06	41,24	28,94	32	69,48	59,40	53,98
40	116,03	86,14	71,52	40	85,41	78,13	66,55
48	123,38	96,99	75,58	48	93,09	78,62	65,14
56	129,86	99,10	75,40	56	103,10	91,09	65,97
64	54,58	43,28		64	180,31	145,58	

(a) Heat Transfer App - App. Size 1000 x 1000

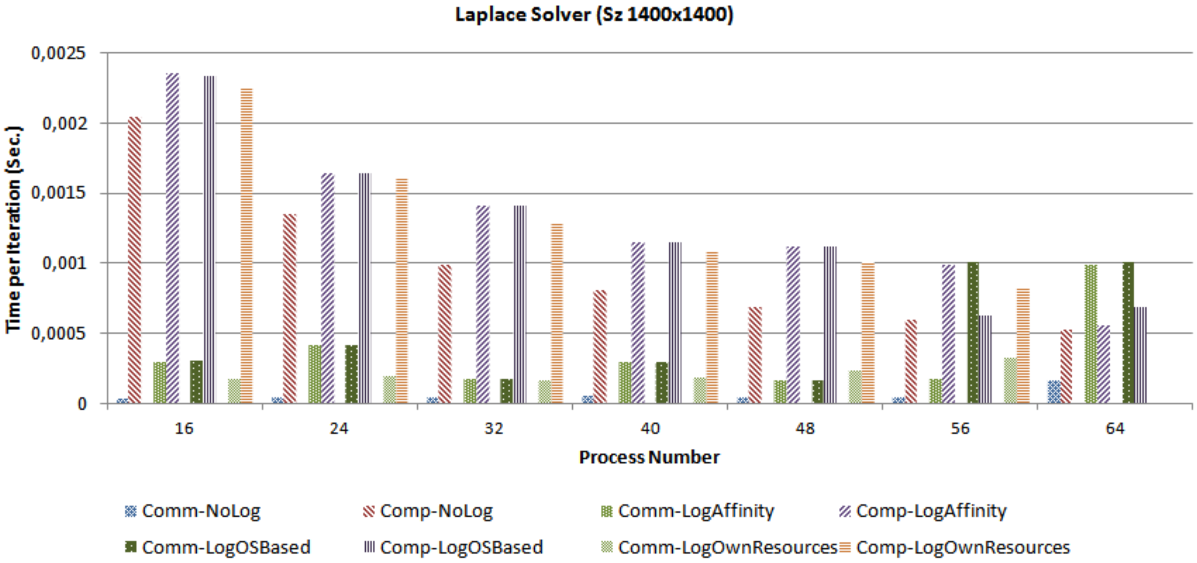
Figure 5.8: Characterization results and Overhead Analysis of Applications using Message Logging.

3. With message logging using all available cores in each node without giving affinity to each logger thread (Comm-LogOSBased and Comp-LogOSBased).

4. With message logging saving one core per node and assigning all logger threads to the core not used by application processes (Comm-LogOwnResources and Comp-LogOwnResources).

We have used a fill-up mapping for the executions without message logging and with message logging using shared resources. For experiments where we assign own resources for the message logging tasks we have use a fill-up mapping policy also for the application processes, but saving the last core of each node for the message logging tasks.

With the purpose of measuring the communication and computation times of each application, we have inserted a barrier (MPI_Barrier) that allows us to properly measure them. The tables of Figure 5.8a and Figure 5.8b show the overhead in percentage introduced by each message logging approach with the barriers and also without them.



Overheads With Barriers				Overheads Without Barriers			
# Process	Ovh-OS-Based	Ovh-Affinity	Ovh-OwnResources	# Process	Ovh-OS-Based	Ovh-Affinity	Ovh-OwnResources
16	39,91	38,79	38,25	16	24,24	24,41	26,25
24	55,79	50,99	50,02	24	40,31	39,60	35,52
32	56,33	37,68	25,74	32	50,28	47,66	42,94
40	75,11	65,36	51,97	40	56,07	57,63	44,82
48	89,70	79,68	65,60	48	80,68	69,55	58,41
56	101,65	82,75	61,83	56	83,45	73,34	52,41
64	87,01	74,10		64	179,03	138,30	

(b) Laplace Equation Solver - App. Size 1400 x 1400

Figure 5.8: Characterization results and Overhead Analysis of Applications using Message Logging. (Cont.)

The executions without barriers are faster than the execution with barriers and we present both overheads in order to prove that the measures taken are consistent when removing them and executing the original versions.

In Figure 5.8a we can observe how the computation times when using the version with own resources is lower. Even when the application becomes communication bound (56 processes) the logging version with own resources behaves better than the other versions. We do not show results of the version with own resources with 64 processes because our test environment has 64 cores, and we have to save 8 cores (1 per node) for the logger threads.

The tables of Figure 5.8a reflect what we have observed when characterizing the application, using message logging with own resources for the logger threads introduces less overhead in almost all cases. With 16 cores without barriers, the version with own resources does not appear to be the best one for a minimum margin, this may be due to the change of mapping from 2 nodes to 3 nodes. At best, we have reduced 25% overhead when comparing the own resources version with the version with shared resources and affinity. We can also observe that when increasing the number of processes without increasing the problem size, the overhead added becomes bigger.

Figure 5.8b shows the execution of the Laplace Solver. As was in the previous experiment, here we can observer how the computation times are lower when using the version with own resources.

The tables of Figure 5.8b reflect again what we have observed when characterizing the application, using message logging with own resources for the logger threads introduces less overhead in almost all cases (except with 16 cores without barriers).At best, we have reduced 20% overhead when comparing the own resources version with the version with shared resources and affinity.

As we have observed, in both applications the computation time of the versions with FT with own resources is lower than the versions with shared resources, but is not equal to the version without message logging. This is because when logging is activated and a process call MPI_Irecv, this process should save the request, re-send the message to its logger thread and free the request when the message was totally received, thus there is an slight increase in computation.

More experiments that validate this methodology are presented in section 6.4.

5.3 Case Study: Increasing Performability of SPMD Applications

In the previous section, we have stated that in order to make efficient resource utilization it is necessary to characterize the message logging impact in combination with the parallel application and the environment. We have decided to analyze a specific parallel paradigm and extend the previous analysis to develop a methodology to make efficient executions of applications while using message logging techniques.

In order to study the impact of message logging techniques we have selected the SPMD paradigm. Processes in this parallel paradigm usually are tightly coupled and there are a lot of message exchanges between them. Thus, this is a very challenging situation for message logging schemes. In this section, we will describe the SPMD applications and a methodology previously developed to characterize executions aiming to improve the efficiency of these applications. Next, we are going to present our methodology [60] to execute SPMD applications, taking into account two conflicting objectives: improving

Performance while giving *Dependability.*

SPMD Applications

When writing parallel applications, it is very important to consider that these applications should made an efficient use of the parallel environment and also an almost linear speedup.

The SPMD paradigm consists in executing the same program in all MPI processes but with a different set of tiles [17]. Hence, for designing our methodology we consider SPMD applications with high synchronicity through tile dependencies and high communication volumes written using MPI for communications.

From now on, when we refer to SPMD applications we mean *Stencil* SPMD codes. In this sense, the SPMD applications that we consider have to accomplish the following characteristics:

- *Static*: which defines the communication pattern and this cannot vary during the execution.
- *Local Communication*: that determines the neighboring communication and it is maintained during all the execution. We are only considering applications with point to point communications.
- *Regular*: the data volume exchanged in each iteration should be the same.
- *N-dimensional*: the applications should be composed by a set of tiles or blocks distributed in one, two or three dimensions.

The parallel applications selected to develop the methodology that will be presented later, are SPMD applications with high synchronicity through tile dependencies and high communication volumes written using MPI for communications. These characteristics give applications a certain level of determinism that allows the usage of a set of techniques to improve performance metrics. We are not including process-level synchronism as BSP (bulk-synchronous parallel) models defined in [82].

There are different kinds of applications of diverse fields that accomplish all the characteristics explained above, and also applications kernels such as the NAS parallel benchmark suite in the BT, MG, SP [3]. Stencil SPMD codes are usually found in computer simulations, e.g computational fluid dynamics, heat transfer, Laplace models, wave equations, Jacobi solver, etc.

We have presented the main characteristics of the SPMD applications that are part of our analysis, now we focus on showing the main challenges when executing these applications on multicore environments.

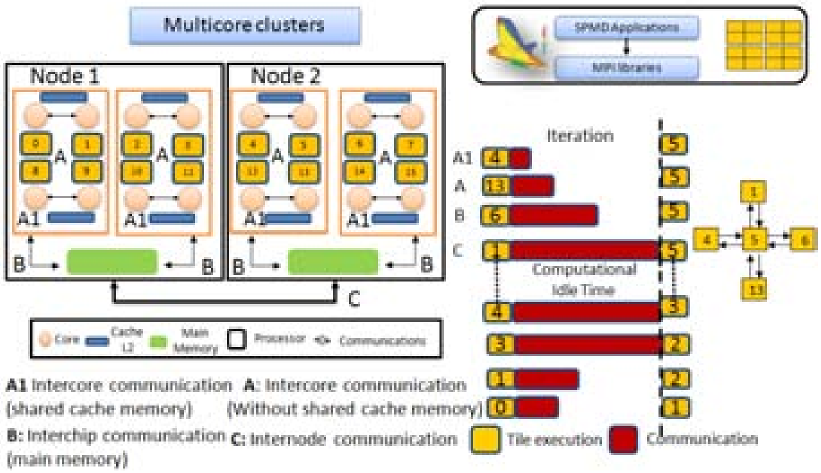


Figure 5.9: SPMD application and communication effects on multicore clusters.

Methodology for Efficient Execution of SPMD Applications

A methodology to write SPMD applications obtaining maximum speedup for a defined level of efficiency was presented in [65]. This methodology has been used as basis to create the methodology described later in this section, which aims for efficient executions with fault tolerant support.

Most of MPI SPMD applications consists in executing the same program in all MPI processes, but using a different set of tiles to compute and communicate. These tiles need to exchange information with neighboring tiles during a set of iterations. For this reason, when SPMD applications are mapped into multicore clusters, code writers must consider the communication heterogeneity and how these can affect the performance specially efficiency and speedup. As shown in Figure 5.9, there are idle times generated by communication links, (e.g. core 1 communicating from node 1 with core 5 of node 2 through the internode link).

The example illustrated in Figure 5.9 shows how an SPMD application may be executed on a multicore cluster with different communication levels. In this case, tiles are computed in a similar time due to the homogeneity inside the cores but the tiles communication times can be totally different because they can be performed using different communication paths (Intercore, Interchip, Internode). In some cases, the differences between each link for a defined packet size can include an order of magnitude of up to one and a half in latency. These variances are translated into inefficiency, which decreases performance metrics.

Taking into account the aforementioned problems, it is very important to manage the communication imbalance of the multicore architecture because it can create dramatic inefficiencies in the parallel region that will be repeated during all the applications'

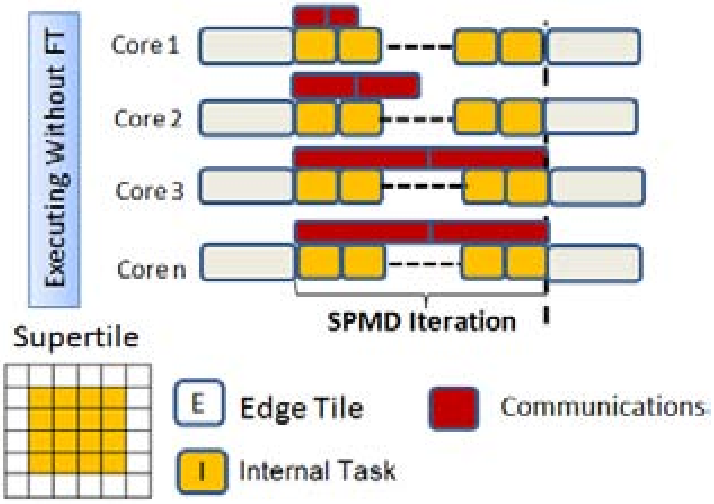


Figure 5.10: SPMD application executed applying the Supertile Concept.

iterations and, as we have seen, even more when we apply fault tolerance support.

In this sense, when we execute a SPMD application without a proper politic it may bring about performance degradation (specially in speedup and CPU efficiency) and imbalance problems [49]. In [65] has been presented a method to manage inefficiencies caused by communication latencies by knowing the characteristics of applications (e.g. communication and computation ratio) and by applying the *supertile* definition [66]. A *supertile* is a structure composed by internal and edge tiles with the aim of allowing us to compute the edge tiles and then to overlap the edge communications with the internal computation in SPMD applications.

The problem of finding the optimal *supertile* size is formulated as an analytical problem, in which the ratio between computation and communication of the tile has to be found with the objective of determining the ideal size that maintains a close relationship between speedup and efficiency. An execution of an SPMD application tuned with the methodology presented on [65] is shown in Figure 5.10.

This methodology is organized in four phases: characterization, tile distribution model, mapping and scheduling. The main objective of this method is to find the maximum speedup while the efficiency is maintained over a defined threshold. Figure 5.11 shows how this methodology is able to improve the efficiency and speedup of parallel applications, in this case a defined efficiency of 85% is being considered. Now, rises the next question: How is affected a tuned application when fault tolerance support is added and the application has to share resources with a fault tolerant middleware?

Later in this section, we will show how we have improved this methodology in order to consider also the effects of fault tolerance tasks and how we can hide the impact of these

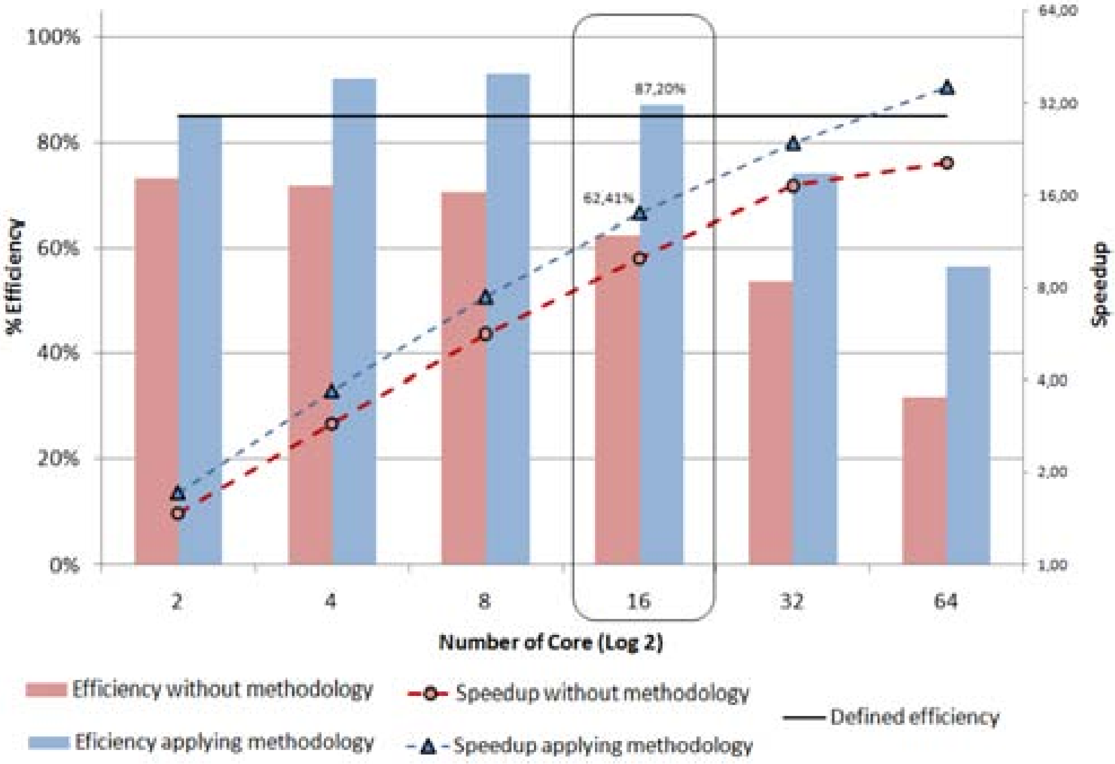


Figure 5.11: Heat Transfer Application. Efficiency and Speedup improvement.

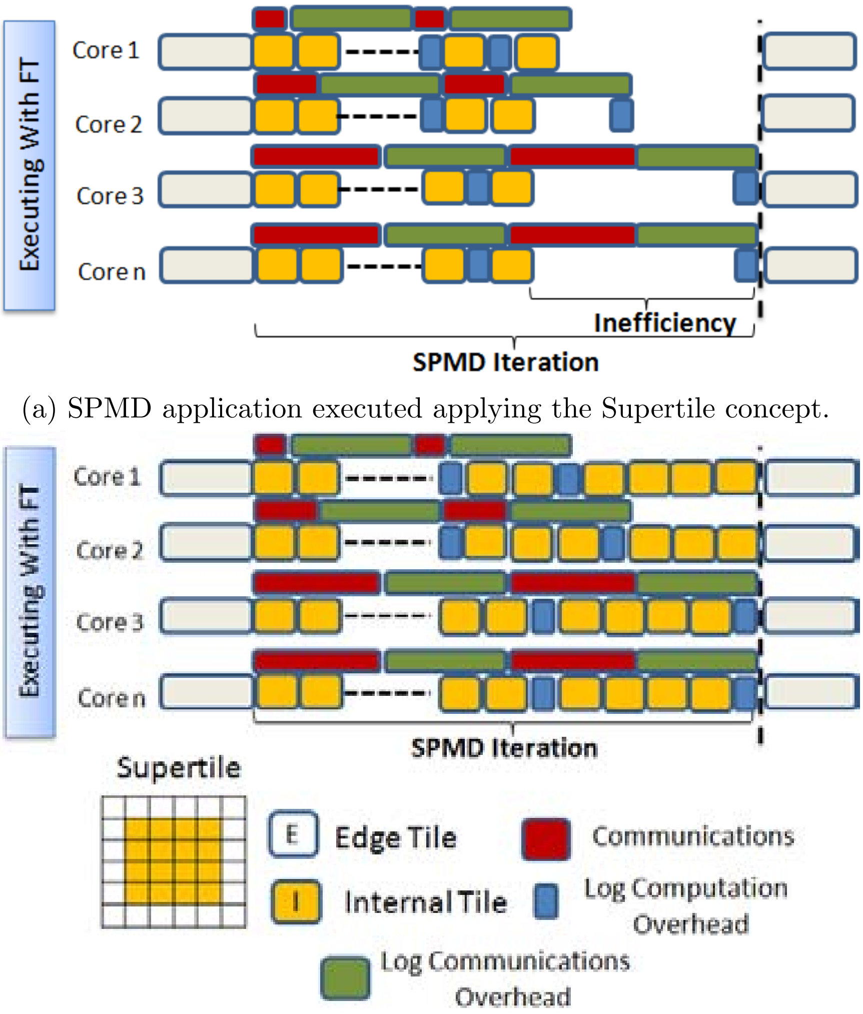
tasks in order to find the maximum speedup using an efficiency threshold. We will also explain in detail the characterization phase (and all the other phases) and how it has been adapted in order to consider the message logging tasks impact.

Methodology to Increase Performability of SPMD Applications

Considering the previous explained methodology, we have focused on using the information that can be extracted in order to adapt and configure applications executions using fault tolerance support making an efficient resource utilization.

The methodology recently presented allows us to balance two very important performance metrics: Speedup and CPU efficiency. However, it is currently important to consider node failure probability since the mean time between failures in computer clusters has become lower [13]. Then, we cannot avoid the fact that running an application without a fault tolerance approach has a high probability of failing and stopping when using many computational resources or when it has been executing for several hours.

Now we are going to describe in detail our methodology [60] to execute SPMD applications, taking into account two conflicting objectives: improving *Performance* while giving *Dependability*. When protecting applications transparently with a fault tolerance



(a) SPMD application executed applying the Supertile concept.

(b) SPMD application with ideal overlap between computations and communications.

Figure 5.12: SPMD Application Behavior using Pessimistic Receiver-based Message Logging

technique (using the available system resources) we should pay an extra overhead that seriously affects the performance. In this sense, all initial tuning may be in vain when we give fault tolerance support without considering the influence of the overhead added on applications (Figure 5.12a). It is important to highlight that we have selected the SPMD paradigm as case study because it represents one of the worst scenarios for message logging techniques, since there are a lot of message transmissions and processes are tightly coupled.

For this reason, the objective of our methodology is to achieve maximum speedup, while considering a defined efficiency threshold, but using a transparent fault tolerance approach. Our methodology is based on the one presented in [65], and it allows us to manage the overheads of a pessimistic logging technique. This is done using an overlapping technique, where the overhead is overlapped with application computation obtaining a considerable improvement in the Performability. Hence, the update of the model allows us to find the ideal number of computation cores and ideal number of tiles which have to be assigned for achieving our objective, taking into account the impact of the fault tolerance (Figure 5.12b).

The main phases of our methodology are described in detail in the next subsections.

5.3.1 Characterization

The main objective of this phase is to evaluate the execution environment taking into account computation and communication times, and also make an analysis of the SPMD application that is going to be executed. The obtained data will be used in the next phase (Distribution Model) to calculate the appropriate number of computational cores and the *supertile* size.

The three main aspects that are considered during the characterization phase are: application parameters, execution environment parameters and the desired performance parameters. When obtaining these parameters, we should consider that we are going to use a message logging approach that will introduce overhead in communications and computations. Thus, when executing each characterization step we are using our modified MPI library that carry on the pessimistic receiver-based message logging of RADIC.

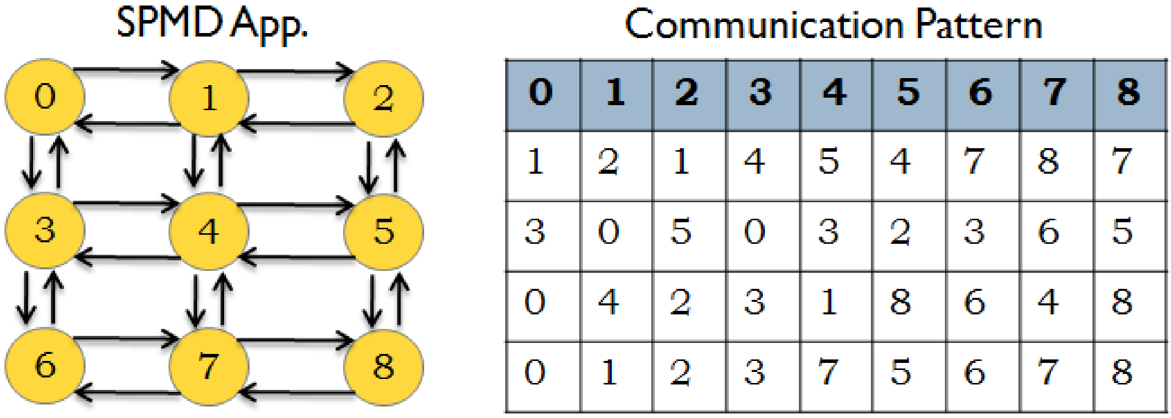
Application Parameters

Our methodology proposes to first discover the set of application parameters that could explain the behavior of the SPMD application. We extract parameters that will be representative when selecting the mapping of message logging tasks and application processes.

First we need to extract the problem size and number of iterations from the application. Then we have to measure the times spent in computing and transmitting a tile. These times are going to be used to calculate the computation-communication ratio which is the key to determine the ideal *supertile* size.

The next step is to find the communication pattern of the application. This pattern will tell us how many neighbors communications are done during each iteration. In order to determine the communication pattern we have used the PAS2P (Parallel Application Signature for Performance Prediction) tool [85]. This tool extracts the main phases of a parallel application by collecting and analyzing computation and communication events.

The information about the communication pattern is collected and used as input for


SPMD App.

Communication Pattern

0	1	2	3	4	5	6	7	8
1	2	1	4	5	4	7	8	7
3	0	5	0	3	2	3	6	5
0	4	2	3	1	8	6	4	8
0	1	2	3	7	5	6	7	8

Figure 5.13: SPMD Communication Pattern Example.

the network characterization module. This module is part of the characterization tool, and it will use this pattern to simulated the behavior of the MPI messages. In Figure 5.13, we show the communication pattern for a Heat Transfer application with 4 neighbors. We should highlight that we only obtain the application communication pattern and when using message logging each communication will be affected since messages should be logged in a different node, but the communication pattern of the application will not change.

Now that we have determined the application parameters we can proceed with the environment characterization, using the obtained parameters to evaluate the communication and computation steps.

Execution Environment

If we want to achieve maximum speedup for a defined efficiency threshold while using a message logging technique, we should make a correct distribution of tiles among the computational cores. If we want to avoid communication bound or computation bound applications we should determine the number of tiles that allows us to fully overlap communication with computations. However, it is very important to consider how the message logging technique affects communications and computations in order to avoid the situation showed on Figure 5.12a. Considering this, when characterizing the execution environment we are running the fault tolerance mechanisms as part of this environment.

In order to evaluate the execution environment, we first analyze and characterize the communication times according to the communication pattern obtained before. After this, we evaluate the computation process so then we can obtain the communication-computation ratio of tiles for a determined environment.

To characterize the hierarchical communication system of a multicore cluster, first

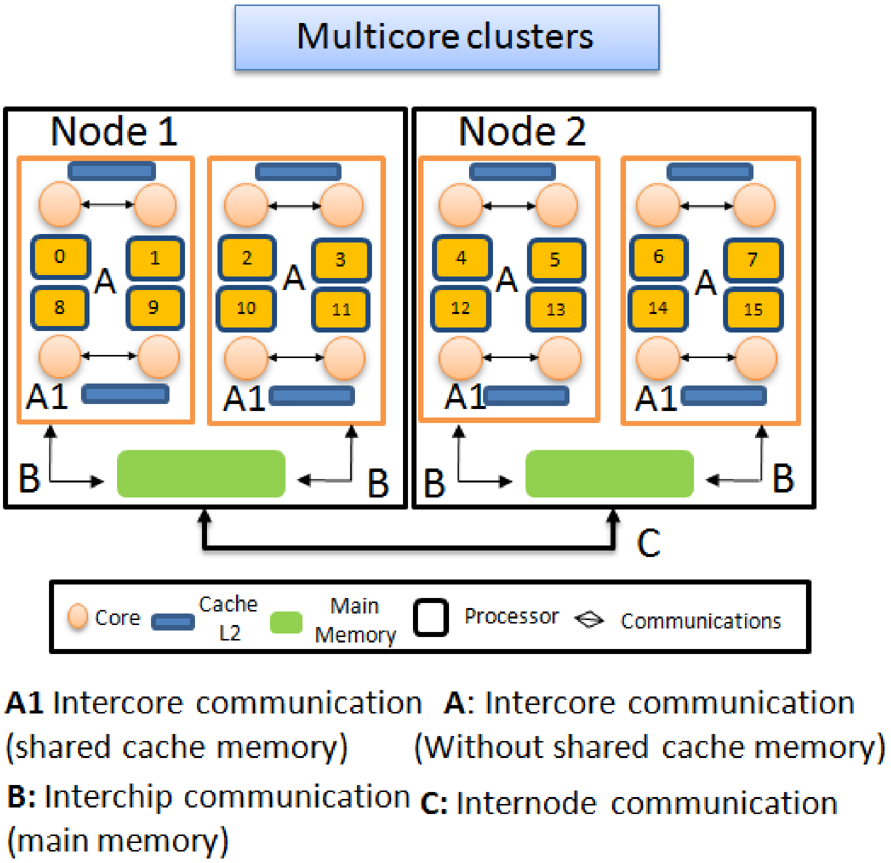

Multicore clusters
Node 1
Node 2
0
1
8
9
2
3
10
11
4
5
12
13
6
7
14
15
A
A1
B
C
Core
Cache L2
Main Memory
Processor
Communications
A1 Intercore communication (shared cache memory)
A: Intercore communication (Without shared cache memory)
B: Interchip communication (main memory)
C: Internode communication

Figure 5.14: Hierarchical Communication levels of a multicore Cluster.

is needed some information about the processors architecture and cache levels. Figure 5.14 shows an example of multicore (2 QuadCores per socket) cluster with different communication levels. Communications between cores inside the same processor can be made without using shared cache memory (A) or using it (A1); communication between cores that belong to different processors are made through main memory (B) and communications between cores that belong to different nodes are made through the available local network.

Once we have determined the different communication levels, the next step is to analyze each of these levels by using a tool for obtaining the different latencies. Several tools to measure communication latency and bandwidth are available, such as NetPipe [81]. These tools collect measures by using Ping Pong techniques. However, Ping Pong tests do not represent the real communication pattern of SPMD applications, since they are designed to communicated with different number of neighbors. Thus, this causes an increase in latency when the system is under full use.

In order to characterize the different communication levels we have developed a characterization tool that allows us to obtain latency times using the real communication pattern of the SPMD application with different number of processes. This tool was written in MPI and its algorithm can be observed in Algorithm 5.1.

```
CommCharacterization(Processes n)
  Logic_Process_Distribution(n)
  Set_Process_Affinity(getPID)
  /*Define the communication topology of
  the application (1D, 2D, ...)*/
  Topology_Creation(Dimension)
  Neighbors_list = Determine_neighbors()
    for message=8B to 16MB
      for iteration=1 to Iteration_Number
        foreach Neighbor in Neighbors_list
          Isend(message, ..., Neighbor, ..., request_send)
          Irecv(message, ..., Neighbor, ... request_recv)
          Collect_Times()
        end foreach
      end for
    end for
    Report_generation()
 end CommCharacterization
```

Algorithm 5.1: Communication Characterization Tool.

Algorithm 5.1 starts by distributing the processes according to the topology of the application (1D, 2D, etc) in order to have one process per core with its neighbors defined from the beginning. In order to ensure that processes are not migrating from one core to another, we use process affinity. However, in case that there are available cores for fault tolerance tasks, this affinity also could be changed. Then we create the topology desired according to communication pattern obtained previously with the PAS2P tool. Once the neighbors are defined we start the iterative process where we send and receive messages of different sizes tofrom each neighbor and collect the times. We repeat for several iterations the measurement procedure in order to obtain reliable times. This characterization tool could be used with different configurations: without message logging, with pessimistic receiver-based message logging or with another fault tolerant strategy.

Once all the measures have been obtained, this information is processed in order to filter the communication time for each communication level (intercore, interchip, internode). An example of how the communication times are affected by the logging operation can be seen in Figure 5.15. The increase in the communication time is due to the retransmission of every received message to another node, even intranode messages will become internode.

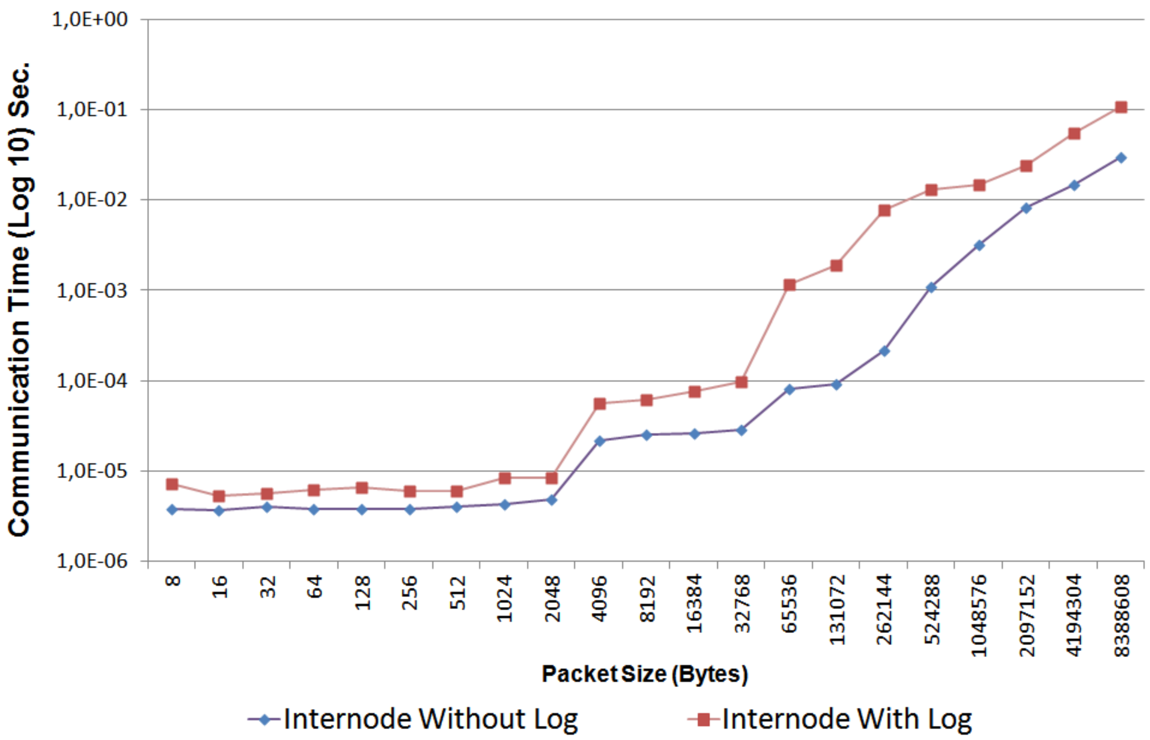


Figure 5.15: Network characterization of the Parallel System.

Once the communication times have been calculated, we should proceed to calculate the computation times of the SPMD application. In order to do this, we instrument the SPMD application for observing the computation function that should be executed for each tile. In order to obtain the computation time of a tile, we assign a set of tiles to each core and measure times averaging total time spent by the number of tiles executed.

Similarly, we should obtain the communication volume of tiles so we can calculate the communication-computation ratio ($\lambda ft(\rho, \omega)$, where ρ indicates the communication link used, and ω indicates direction e.g. up, right, left or down in a four communication pattern).

In order to properly characterize the computation times while the message logging technique is being used, we have combined the kernel that make the computation with our Network Characterization Tool.

Algorithm 5.2 shows the operation of our characterization tool. We give a set of tiles as input that is distributed among the number of processes that is being used. We have one thread that is in charge of communications of tiles and the main thread that computes the tiles, since the parallel applications that we consider here are written in that way. The computation thread first blocks a semaphore and computes the *supertile* edges. After computing the edges, the thread unblocks the semaphore and compute the internal tiles. The communication thread blocks the semaphore when it is available and sends the necessary tiles to neighbors using non-blocking MPI communications and when the communication is finished, unblocks the semaphore so in the next iteration the main thread can compute the edges. By doing things in this way, we are able to determine how the computation times are affected by the message logging technique, since the logger threads also consume CPU cycles.

It is important note that when using non-blocking MPI receptions, messages are logged effectively when the corresponding wait operation is executed. This is because just after the wait we guarantee that the message has been fully received and it can be logged.

Now that we have calculated communication and computation times for tiles, we will proceed to explain analytically the characterization stage.

As a *supertile* is a structure composed of internal and edge tiles (Figure 5.10), we can divide the computational time of a tile in two: the internal tile (Cpt_{int}) and the edge tile computation (Cpt_{ed}). The internal tile is divided in the original computation of a tile[1] (Cpt_{tile}) plus a piece of the overhead added by the logger threads ($FTCpt_{int}$) in the protection step of the FT approach.

Then, the internal computation is represented in Equation 5.1 (Figure 5.12b).

$$Cpt_{int} = Cpt_{tile} + FTCpt_{int} \tag{5.1}$$

The edge tile computation without FT (Cpt_{tile_ed}) needs to consider the time spent in packing and unpacking the sent and received tiles, plus the time of computing them (Equation 5.2). However, when calculating the edge tile computation time using FT (Cpt_{ed}), we should consider the overhead caused by the logger process ($FTCpt_{ed}$), as it can be detailed in Equation 5.3.

$$Cpt_{tile\_ed} = Pack\_Unpack + Cpt_{tile} \tag{5.2}$$

$$Cpt_{ed} = Cpt_{tile\_ed} + FTCpt_{ed} \tag{5.3}$$

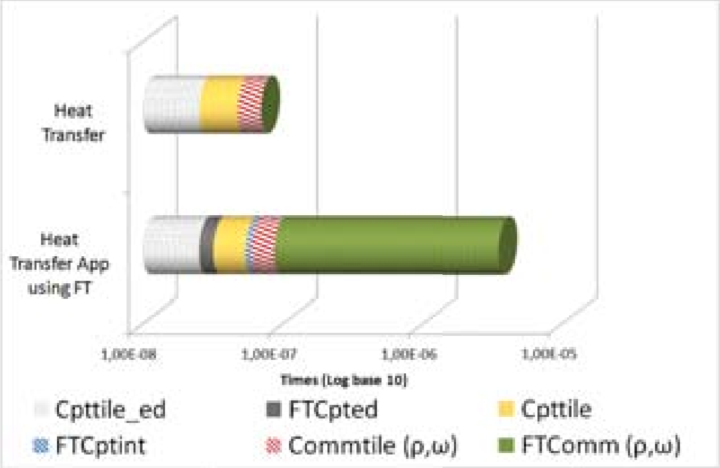
The communication time is measured as the communication of a tile ($Comm_{tile}(\rho \quad)$), plus the overhead of the log operation ($FTComm(\rho \quad)$), hence, the tile communication applying FT is calculated with Eq 5.4 (Figure 5.15).

[1] The Cpt_{tile} is the original tile computation used on the method without fault tolerance approach [66]

```
CompCharacterization(Processes, Tiles)
  Logic_Process_Distribution(Processes)
  Set_Process_Affinity(getPID)
  /*Define the communication topology of
  the application (1D, 2D, ...)*/
  Supertile = Tiles/Processes
  Topology_Creation(Dimension)
  Neighbors_list = Determine_neighbors()
  Block(Semaphore)
  CreateThread(CommThread, Neighbors_list, Supertile)
  for iteration =1 Iteration_Number
    Block(Semaphore)
    Compute_edges(Supertile.edge)
    Unblock(Semaphore)
    Compute_internals(Supertile.internals)
    Collect_Times(Comp)
  end for
  Thread_Join(CommThread)
  Report_generation()
end CommCharacterization

Function CommThread(Neighbors_list, Supertile)
  for iteration=1 to Iteration_Number
    Block(Semaphore)
    foreach Neighbor in Neighbors_list
      Isend(Supertile.edge, ..., Neighbor, ..., request_send)
      Irecv(Supertile.edgeNew, ..., Neighbor, ... request_recv)
      Collect_Times(Comm)
      Supertile.edge=Supertile.edgeNew
    end foreach
    Unblock(Semaphore)
  end for
end CommThread
```

Algorithm 5.2: Computation Characterization Tool.



	Border Times			Internal Times			Communication Times		
	Cpt tile_ed	FTCpt ed	Cpt ed	Cpt tile	FTCpt int	Cpt int	Comm tile (ρ,ω)	FTComm (ρ,ω)	CommT
Heat Transfer	2,49E-08	0	2,49E-08	2,21E-08	0	2,21E-08	3,79E-06	0	3,79E-06
Heat Transfer App using FT	2,49E-08	6,63E-09	3,15E-08	2,21E-08	5,08E-09	2,72E-08	3,79E-06	3,36E-06	7,15E-06

Figure 5.16: Computation and Communication characterization of the Heat Transfer Application.

$$CommT(\rho, \omega) = Comm_{tile}(\rho, \omega) + FTComm(\rho, \omega) \tag{5.4}$$

To calculate the communication-computation ratio, we only consider the internal computational time (Cpt_{int}) as this is the only computational time which can be overlapped with communications (Equation 5.5).

$$\lambda ft(\rho, \omega) = CommT(\rho, \omega)/Cpt_{int} \tag{5.5}$$

As can be detailed in Figure 5.16, the overhead added in a message when we apply the message logging technique has a considerable impact, which has to be considered for the model precision. As the logger threads of the protectors also consume CPU cycles, we cannot avoid that fact when characterizing the computation. Considering this, our characterization tool is designed to extract the computation time of each tile and the overhead added by the logger. As can be evidenced in Figure 5.16, there are two overheads which have to be managed when overlapping ($FTCpt_{int}, FTComm$).

Performance Parameters

In order to make use of the model we need to define an efficiency threshold ($effic$) that will be used to calculate the ideal *supertile* size. This efficiency threshold is defined by the user and it will be used in the analytical model to calculate the number of cores and *supertile* size that achieves the maximum speedup considering the efficiency level desired.

5.3.2 Distribution Model

This step of the methodology focuses on finding the ideal size of the *supertile* that will allow us to avoid communication bound or computation bound executions when using a message logging approach. In the next paragraphs, we will explain the analytical process of data distribution using the values obtained in the characterization phase.

By taking into account overheads of the message logging technique into SPMD applications, in both computations and communications, we obtain the next values for a tile: Cpt_{ed}, Cpt_{int} and $CommT$. Once we have characterized our environment and obtained the size of the *supertile* and the communication-computation ratio (Equation 5.5), we can obtain the optimal number of cores to be used in order to maintain the speedup under a defined efficiency threshold.

The analytical model for improving performability begins by determining the ideal number of cores ($Ncores$) that allows us to find the maximum speedup under a defined efficiency, but managing the log effects of the fault tolerance technique (Eq.5.6). However, this equation depends on the problem size (M^n), where n is the application dimension (e.g 1,2,3, etc), and ideal supertile size (K^n).

$$Ncores = M^n \;\; K^n \tag{5.6}$$

Furthermore, to obtain the value of K, we have to start by Equation 5.7, which represents the execution time of the SPMD application using the overlapping strategy. We can first calculate the edge computation time $EdComp_i$ (Equation 5.8, where ST is the *supertile* size), then we add the maximum value between internal tile computation $IntComp_i$ (Equation 5.9) and edge tile communication $EdComm_i$ (Equation 5.10), applying the overlapping strategy. This process will be repeated for a set of iterations $iter$. All these values depend on the log effects in computation and communication that have been obtained in the characterization phase. It is important to consider that Equation 5.8, Equation 5.9 and Equation 5.10 are in function of Cpt_{ed}, Cpt_{int} and $CommT$ and these were calculated in the characterization phase while using the message logging approach.

$$Tex_i = \sum_{i=1}^{iter} (EdComp_i + Max(IntComp_i \;\; Edcomm_i)) \tag{5.7}$$

$$EdComp_i = (ST - (K-2)^n) * Cpt_{ed} \tag{5.8}$$

$$IntComp_i = (K-2)^n * Cpt_{int} \tag{5.9}$$

$$EdComm_i = K^{n-1} * Max(CommT(\rho \quad)) \tag{5.10}$$

Hence, Equation 5.11 represents the ideal overlapping that allows us to obtain the maximum speedup (considering the protection time), while the efficiency $effic$ is maintained over a defined threshold. Therefore, we start from an initial condition, where the edge communication time with log operation is bigger than the internal computation time divided by the efficiency. This division represents the maximum inefficiency allowed by the model. However, Equation 5.11 has to consider the constraint defined in Equation 5.12 where $Edcomm_i$ can be bigger than $IntComp_i$ over the defined efficiency (Equation 5.11), but the $Edcomm_i$ has to be lower than the $IntComp_i$ without any efficiency definition.

$$K^{n-1} * Max(CommT(\rho \quad)) \geq \frac{(K-2)^n * Cpt_{int}}{effic} \tag{5.11}$$

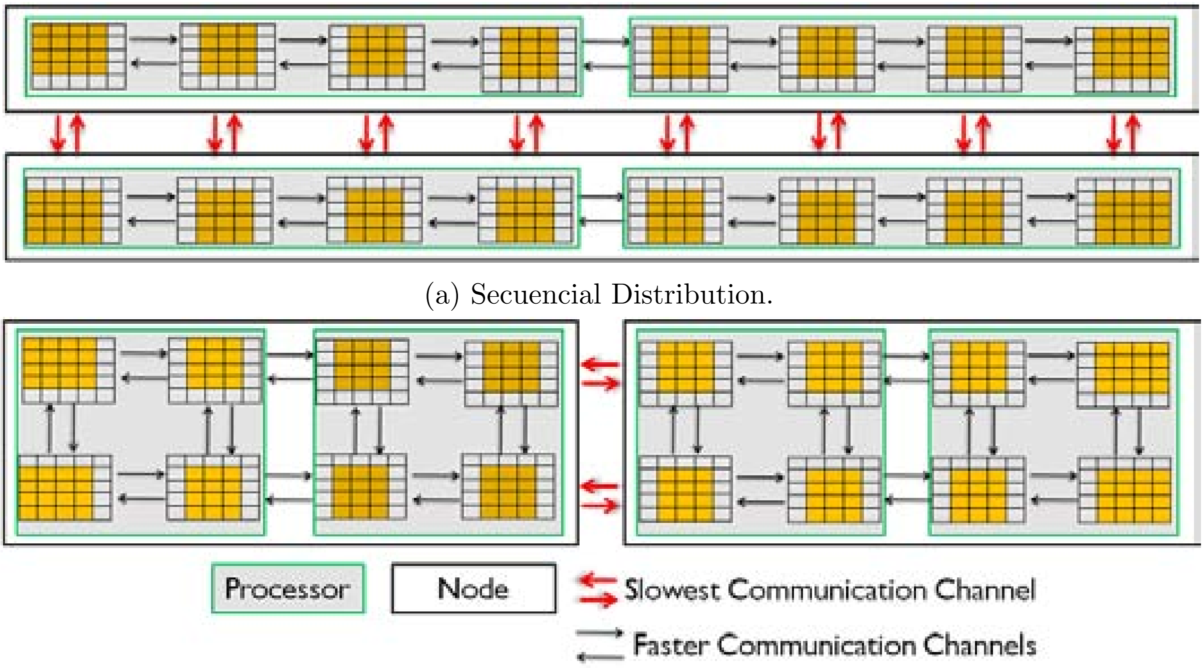
$$K^{n-1} * Max(CommT(\rho \quad)) \leq (K-2)^n * Cpt_{int} \tag{5.12}$$

However, the edge communications are in function of the $CommT$. For this reason, we need to equalize the equation in function of Cpt_{int}. This is achieved using Equation 5.5, where we can isolate the $CommT$ in function of the Cpt_{int} multiplied by $\lambda ft(\rho \quad)$. Having both internal computation and edge communication in function of Cpt_{int}, the next step is to replace this value in Equation 5.11 and we obtain the Equation 5.13.

$$K^{(n-1)} * max(\lambda ft(\rho \quad) * Cpt_{int}) = ((K-2)^n \;\; effic) * Cpt_{int} \tag{5.13}$$

To find the value of K, we equal the Equation 5.13 to zero and we obtain a quadratic equation in the case of using an SPMD application with 2 dimension (Eq.5.14). The solution obtained has to be replaced in Eq 5.11 and Eq 5.12 with the aim of validating if the K value accomplishes the constraint defined.

$$K^2 - 4 * K - effic * \lambda ft(\rho \quad) * K + 4 = 0 \tag{5.14}$$



(a) Secuencial Distribution.

(b) Block Group Distribution.

Figure 5.17: Logical Distribution of Supertiles among cores considering two nodes with 2 quad-core processors each.

The next step is to calculate the ideal number of cores (Equation 5.6), taking into account the impact of the fault tolerance strategy used. Also, Equation 5.6 allows us to determine the ideal ST size for a specific number of cores with the aim of checking how our method scales considering weak scalability.

5.3.3 Mapping

The aim of this phase is to allocate each *supertile* into each core with the aim of minimizing the communication effects. We have designed a strategy to locate MPI processes in the cores where the *supertiles* are going to be executed. It is important to highlight that only one *supertile* is going to be executed by core, but also a logger thread will share this resource with it. Our mapping strategy focuses on minimizing the communications effects, this is done by analyzing each communication and minimizing the communications by slower channels.

Our process mapping technique should be able to select the more suitable core to each process and this mapping should be the same that has been used during the characterization phase where we obtained the times of each communication channel. This will avoid variations between the times obtained in the characterization phase and the execution of the application.

The mapping phase is divided in three key parts, which are: the logical distribution of MPI process among cores; the affinity process which attaches each process to one specific core; and the tile division and assignation. We are going to explain each part in the next paragraphs.

Logical Distribution of Processes

During the characterization phase, we have determined the slowest communication channels, thus when we make the logical distribution of processes we focus on minimizing the number of communications using the slowest channel.

Depending on the tile distribution and assignation, the *supertiles* may have a higher number of communications through the slowest channel, and this will cause that the communications made through this channel become even slower.

In Figure 5.17a we show an example of a sequential distribution of tiles among cores, this means that we use a fill-up policy to put the *supertiles* in processes in a consecutive way and attaching each process to cores by using affinity. This way of distributing processes may cause a congestion in the slowest communication channel because we have several communications that use this channel. In this distribution scheme we have more communications that use the slowest channel comparing to a block grouping scheme.

In Figure 5.17a we are considering the utilization of 2 nodes and we have 16 communications through the slowest channel. If we increase the number of nodes, the processes that are executed in nodes that are in the middle will double the number of communications through the slowest channel, making the congestion even worst. Although, the sequential distribution does not affect the model, because if we characterize the network using this distribution and then apply the methodology we will still be able to calculate the communication-computation ratio, however there are other strategies that could minimize the number of communication through the slowest channel reducing congestions.

Figure 5.17b illustrates how we can group the processes and distribute the *supertiles* considering the physical architecture of the environment. We can observe that the number of internal communications increases when using this distribution, but at the same time the number of communications through the slowest channel is reduced. In this case we reduce the number of external communications to 4.

This block distribution strategy will be more efficient as we increase the number of processes. Let us suppose that we want to increase the number of nodes to 32 quad core machines, and that the size of the problem is M^2.

Let us consider that we have the 32 quad core nodes distributed in a mesh of 16 x 16 cores (16 x 2 QuadCore nodes) to distribute the 256 processes. If we distribute the supertiles using sequential distribution we will have the next: $16 * 2$ communications between each pair of nodes in the same horizontal nodes; $(16-1)*32$ vertical communications of internal nodes because the first and the last 16 cores only communicate to one set of 16 cores. In total we have 512 communications using the slowest communication channel.

However, if we use a block logic distribution strategy we will set the nodes in mesh of 4 x 8 QuadCore nodes (16 x 16 cores) we will have the next: $12 * 8$ communications between horizontal node neighbors; $(8-1) * 32$ vertical communications between internal nodes. In total we have 320 communications using the slowest channel. Then, if we use the block logic distribution strategy we reduce the congestion in the slowest network.

In order to increase the speedup of the application, the ideal solution is to assign the minor amount of tiles to each process. However, when we desire a fast and efficient execution, the number of tiles assigned to each process should maintain a close relation between the slower communication and the internal computation time.

Once we have defined the logical distribution of processes, the next step is to apply process affinity. This is done with the information obtained in the logical distribution process.

Process Affinity Model

The aim of the process affinity procedure is to attach each application process to specific physical resources (cores). This is done in order to avoid that processes migrate from one core to another and obtain better performance metrics. By attaching processes to cores, we can easily manage the communications between MPI processes and achieve a more stable SPMD application execution since all communications are controlled by the logical distribution of processes described before.

The affinity procedure is divided in two: the first is the assignation of the *supertiles* and processes to a specific node; and the second is the procedure of attaching each MPI process to a specific core. We first obtain the PID of each MPI process and then we use the sched.h library in order to associate each process to a core.

It is very important to consider that we also have one logger thread running per each application process (running in a different node to save its received messages). We also apply process affinity to each of these logger threads in order to equitably distribute the computation overheads among each core or we can also give own resources to the fault tolerance tasks.

Figure 5.18 shows how we assign each application process and fault tolerance process to specific resources (cores). Each logger thread shares a core with one application process,

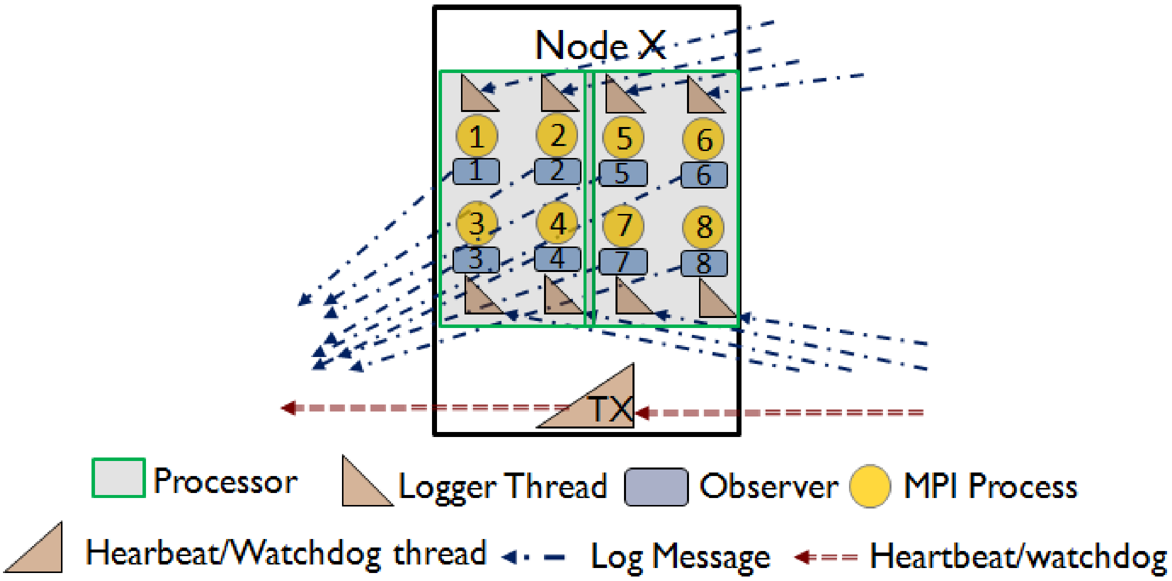


Figure 5.18: Affinity of Application Processes and Fault Tolerance Processes

then we can ensure that overheads are homogeneously distributed among the system.

Division and Distribution of tiles

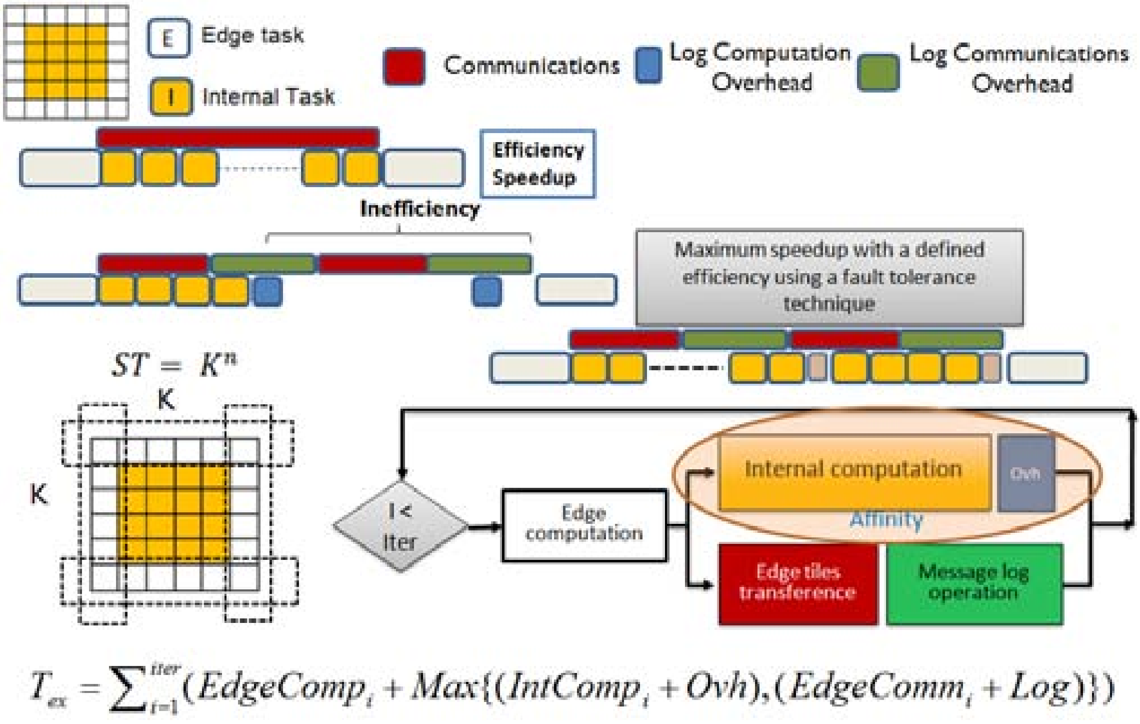
Once we have defined the location of each MPI process, we now should proceed to assign the *supertiles* to each process. The *supertile* size is defined according to the values obtained in the distribution model phase. Also the coordinates assigned to each *supertile* allow us to determine the neighbors according to communication pattern obtained.

When we divide the *supertiles* and distribute the processes, we take into account the physical layout of the parallel architecture.

5.3.4 Scheduling

The last phase of the methodology defines the temporal execution planning of the *supertiles*. This phase is divided in two main parts:

- **Priority assignation**: The edge tiles have the higher execution priority since these tiles should be transmitted to neighbors. The edge tiles are saved into buffers to execute them before the internal tiles. These buffers are updated in each iteration. First, the thread in charge of the computation computes the edge tiles and allow the communication thread to transfer these edge tiles while it continues with the computation of internal, then the internal computation is overlapped with the edge tiles communications.



$$T_{ex} = \sum_{i=1}^{iter} (EdgeComp_i + Max\{(IntComp_i + Ovh),(EdgeComm_i + Log)\})$$

Figure 5.19: SPMD methodology summary.

- **Overlapping Strategy**: Each iteration of the SPMD execution is delimited by the slower communication, so even if some processes have a shorter communication time we should give enough tiles to each process to cover the longest communication. As we are using message logging, we should use a *supertile* size that allows us to cover the logging effects. We have created two threads to carry out the execution. One thread is in charge of computing the internal tiles, while the other communicates the edge tiles. This allows us to hide the message logging overheads by giving more tiles to compute to a processor while edge tiles are being sent and logged.

Figure 5.19 summarizes our methodology. Considering that we have a tuned SPMD application were internal computation is overlapped with edge communications, when we apply a message logging logging technique we introduce inefficiency to the execution. Once we determine the appropriate *supertile* size considering the impact of the message logging technique (using the explained characterization strategy) we can obtain the maximum speedup with a defined efficiency threshold. Usually, logging approaches are combined with an uncoordinated checkpoint approach (such as the one used in RADIC). For calculating our estimations, in this work we do not consider the added overhead that will be caused by the checkpoints.

5.3.5 Performability Framework

Considering the methodology presented, we have designed a framework that allows us to develop performability-aware SPMD applications. This framework allows us to automatize the procedure described previously. In order to ease the analysis and due to the computation homogeneity, we consider that the time spent in uncoordinated checkpoints will be added to the execution time and it will depend exclusively on the application size and the number of checkpoints.

In order to make use of the performability framework with fault tolerance support, the MPI library used should be the one that has RADIC included [59]. Our framework considers that the user code is written in the C language using MPI to carry out communications between processes.

Figure 5.20 shows the flowchart of the performability framework (user inputs are highlighted in green and M^n is the problem size). The user introduces the application inputs, number of iterations (i), the communication code is introduced with medatada that describes the behavior (e.g. number of neighbor communications) and the computation code that details the code that each parallel process should carry out. The user should also introduce the parallel environment, specifying number of cores that each node contains and internode configuration (cache levels, cores that share cache). Using the input, we characterize the environment obtaining the communication and computation time with and without considering the impact of message logging.

Once characterization is over, the distribution model and the analytical model are applied, obtaining the ideal number of computational cores that can be used to obtain the requested efficiency (information about expected execution time with and without fault tolerance support will be available to the users). After confirming the execution with the obtained number of cores($Ncores$) and *supertile* size (ST) the user will have the source code written.

When launching the execution, at first the mapping step takes place and all the work is divided between the number of processes (one process per core). Each process is attached to a core by using an affinity procedure and then two threads are created per process: Computing thread and Communication thread. The computing thread computes the border and then computes the internal tiles. The Communication thread is in charge of transmitting the border values to neighbors, this operation is overlapped with the internal computation. When message logging is being used, the Border Comm. value includes the time of transmitting and logging the edge tiles. The operations of both threads are repeated for the number of iterations (i) of the problem. All RADIC operations are user-transparent, the main difference between whether or not to use fault tolerance resides

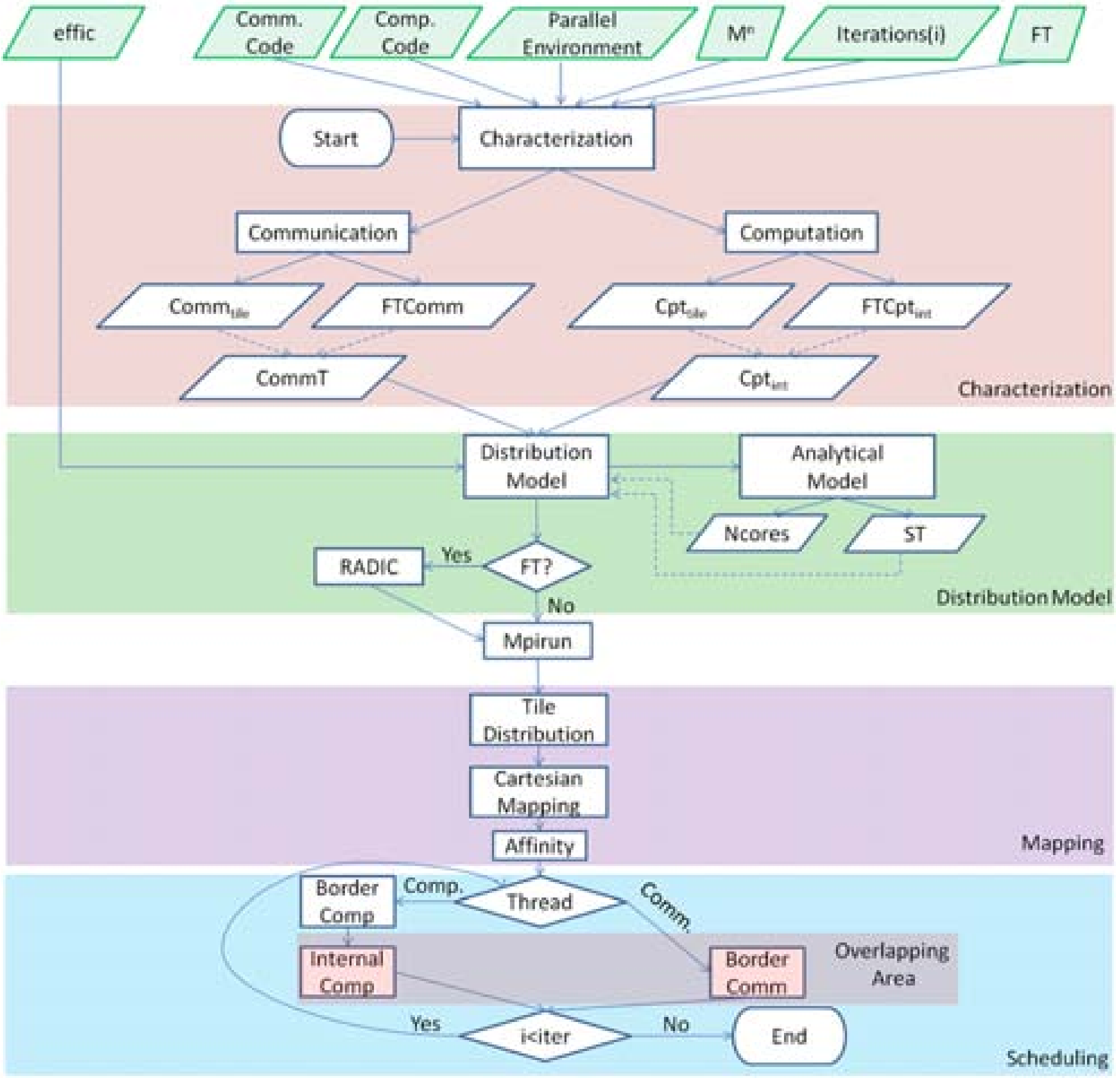


Figure 5.20: Flowchart of the Performability Framework.

in the number of cores and Supertile size.

5.3.6 Validation Example

In order to validate our methodology, we have use a Heat Transfer which is a finite difference application with four communications per iteration. Details about the experimental environment can be found in section 6.1.

Table 5.1 summarizes the theoretical and practical data that allows us to solve Equation 5.14 in order to obtain the supertile size (ST) and the number of cores ($Ncores$) for a defined efficiency ($Effic$). For these executions, we have set a 75% level of desired efficiency. By using the data obtained during the characterization phase, the desired level

Table 5.1: Theoretical and Practical Data of Heat Transfer Application.

Application	Problem Size	Desired *Effic*(%)	Cpt_{int}	*CommT*	λft	ST	*Ncores*	Obtained *Effic*(%)	abs(Error) (%)
Heat Transfer Tuned	1000x1000	75	$2\ 2E^{-8}$	$3\ 79E^{-6}$	172.09	133.03	56	76	1
Heat Transfer Tuned With FT	1000x1000	75	$2\ 71E^{-8}$	$7\ 15E^{-6}$	263.96	201.95	24	80.4	5.4

of efficiency and by solving Equation 5.14 we have obtained a suitable number of cores to be used for each case (Column $Ncores$). Then we have evaluated our predictions by executing the applications with the predicted number of cores obtaining errors that range between 1% and 5.4%.

More experiments that validate this methodology are presented in section 6.5.

5.4 Discussion

In order to meet the performability objective when executing parallel applications on computer clusters, we should consider how the fault tolerance tasks affect applications tuning. In this chapter we have considered a rollback-recovery approach where most of the overhead is caused by a pessimistic logging protocol. We have presented a methodology that focuses on analyzing possible configurations of message logging tasks in order to find the most suitable according to application behavior. This is done by characterizing the parallel application (or a small kernel of it) obtaining the computation and communication times and the disturbance caused by the logging approaches.

As a case study, we have selected the SPMD paradigm and we have applied the previous characterization methodology and proposed a methodology to tune SPMD applications considering the impact of message logging tasks. This methodology allows to obtain a maximum speedup while the efficiency is maintained over a threshold. We have also designed and implemented a framework that allows the usage of this methodology and eases the development of performability-aware SPMD applications.

We can conclude that is becoming extremely important to consider the impact of fault tolerance techniques in parallel executions, since these techniques can unbalance application executions. Our proposal allows us to provide a new feature to applications that have been tuned to be executed efficiently, this feature is resiliency. Using our methodology, users can determine analytically how well an application will run in terms of speedup and efficiency, without doing exhaustive executions in order to obtain the ideal number of processes or computer cores to be used.

Chapter 6

Experimental Results

In this chapter we present the experimental results of the main proposals presented in this thesis. First, we detail the experimental environment in Section 6.1. The description of the experimental environment includes the description of cluster used to evaluate our proposals and the main MPI library used to implement our proposals. Benchmarks and applications used are explained separately in each section.

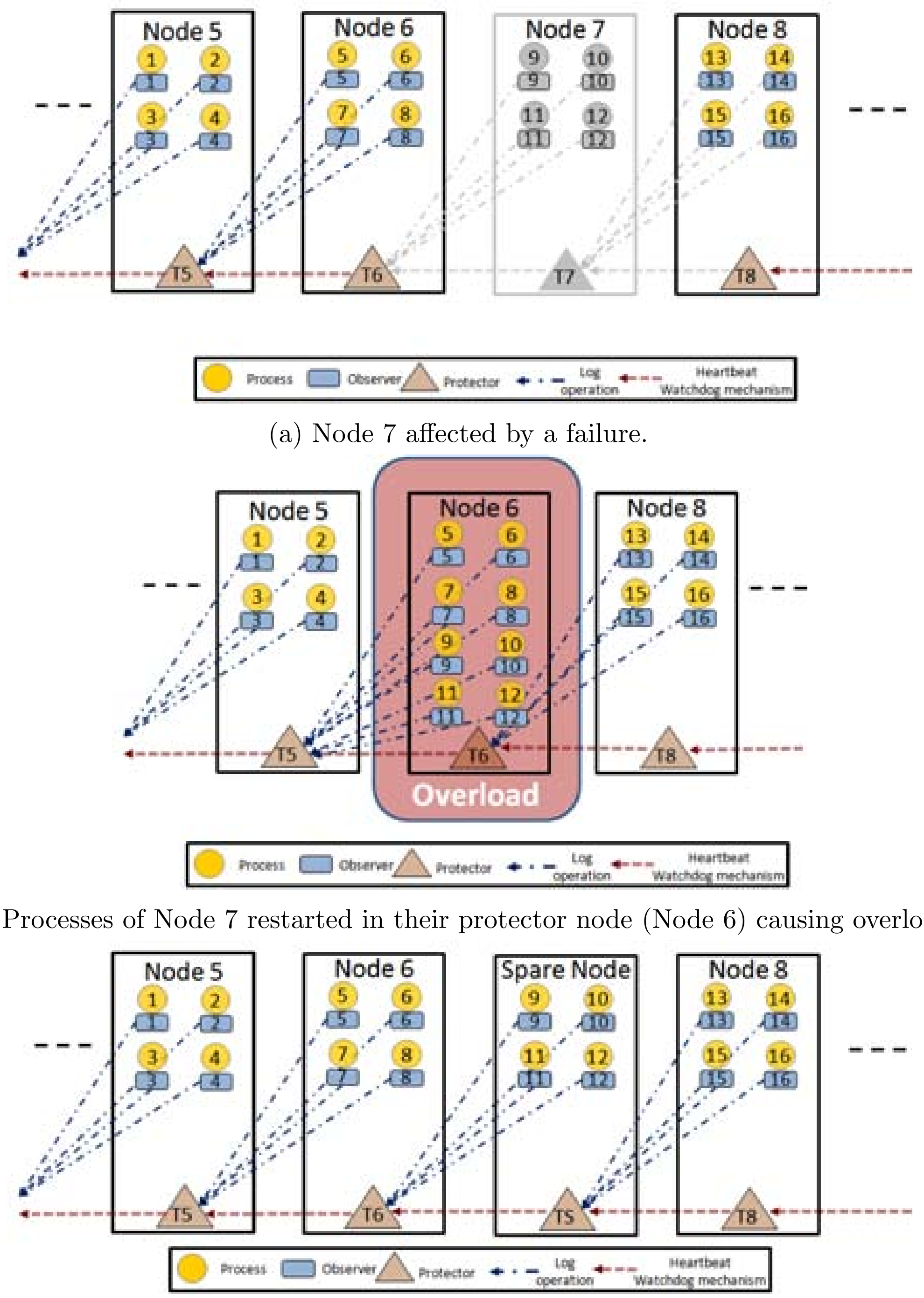
Then, in section 6.2 we present the evaluation and analyze the results obtained when using automatic spare nodes inside the RADIC architecture. Section 6.3 presents the results obtained with the Hybrid Message Pessimistic Logging (HM_{PL}) approach (detailed in section 4.2), and how it benefits parallel executions by lowering the overheads. In section 6.4, we present the evaluation of our methodology to select the most suitable configuration alternative of message logging tasks. Then, section 6.5 presents the evaluation that has been carried out with the methodology to increase performability of SPMD applications.

Finally, the discussion and conclusions of the experiments are presented.

6.1 Experimental Environment

The experiments have been made using a Dell PowerEdge M600 cluster with 8 nodes, each node with 2 quad-core Intel® Xeon® E5430 running at 2.66 GHz. Moreover, each node has 16 GB of main memory and a dual embedded Broadcom® NetXtreme IITM 5708 Gigabit Ethernet is used to interconnect the nodes. All the experiments have been made using only one network interface to process communications and fault tolerance tasks. The operating system used was Red Hat 4.1.1-52.

In order to provide an application transparent FT support, we have included the RADIC architecture and all the implementations made in this thesis inside an Open MPI 1.7. We have considered that in order to give support to MPI applications a good option



(a) Node 7 affected by a failure.

(b) Processes of Node 7 restarted in their protector node (Node 6) causing overload.

(c) Failed processes restarted in a Spare Node.

Figure 6.1: Failure Recovery in RADIC with and without Spare Nodes.

is to act as a middleware between the MPI application and the MPI library. This allows us to execute any MPI application without modifying the source code. More details about the implementation of RADIC inside Open MPI could be found in subsection 2.3.4.

6.2 Operation of RADIC Fault Tolerant Architecture

Fault tolerance mechanisms must try to reduce the added overhead during failure-free executions and avoid system degradation in presence of failures. In subsection 2.3.3 we have discussed the operation modes of the RADIC architecture. RADIC allows the successful completion of parallel applications even in presence of failures, without using extra resources. However, it is important to avoid system degradation and reduce the overhead when handling node failures. If failed nodes are replaced automatically by spare nodes, parallel applications will not only end correctly but also will avoid performance degradation due to loss of computational resources.

6.2.1 Design and inclusion of Spare Nodes in RADIC

The spare node management mechanism that we have included in RADIC is not restricted to avoid performance lost, but also we propose a mechanism for automatically select spare nodes and include them on the parallel environment domain without user intervention. By doing the spare nodes management transparently and automatically, we are able to minimize the MTTR.

Two main data structures are used in RADIC during recovery: the RADICTable and the SpareTable. As we may recall, the RADICTable has one entry by application process and each row contains: process id, URI (Uniform Resource Identifier), URI of process protector, the Receive Sequence Number (RSN) and Send Sequence Number (SSN). The SpareTable contains the spare Id, the address of the spare and the status (free or busy). The RADICTable and SpareTable are replicated among all protectors.

When using RADIC without spare nodes, failed processes (Figure 6.1a) are restarted in their protectors (Figure 6.1b) causing overload. If an observer tries to reach a relocated failed process, it will take a look at its RADICTable to find the old protector of the failed process (T6). Then, the observer will ask about that process. The old protector (T6) will say that it is no longer protecting such a process, and will point who is the new protector (T5 in Figure 6.1b).

When using spare nodes and a failure occurs, the protector of the faulty processes consults its SpareTable, and if there is one available spare node (status $\bar{\text{f}}$ree) the protector contacts the spare and asks if it can adopt the failed processes. In order to solve race conditions, the consulted spare will answer protectors using a FIFO protocol. Protectors will update their local SpareTable according to the answers of the spare nodes. If a consulted spare node is *free*, the protector of the failed processes transfer the checkpoint and message log to the spare node and failed processes are restarted (Figure 6.1c). This spare node will change its status to *busy*. Spare nodes replies with *busy* when they do no have enough resources to hold all failed processes.

Considering Figure 6.1c, if process 1 from Node 5 wants to reach process 9, observer 1 will ask T6 about process 9. T6 will point that process 9 is residing now in a spare node. Then observer 1 will tell T5 to update its RADICTable and its SpareTable and process 1 will finally contact process 9. The process described above is distributed and decentralized, and each process will do it only when it is strictly necessary, avoiding collective communications. The protector TS has an initial copy of the RADICTable and the SpareTable, and these tables will be updated on demand.

By using spare node support, the scalability of RADIC architecture is reinforced, since degradation is avoided.

Taking into account the inclusion of RADIC inside the Open MPI library, the main problem when restarting a process in another node is that we need an ORTE daemon running in that node to adopt the new process as a child. Moreover, all future communications with restarted processes needs to be redirected to their new location. For that reason, ORTE daemons are launched even in spare nodes, but no application process is launched on it until it is required as a spare node.

An additional problem that must be addressed is that a sender observer must not consider as a failure the lack of communication with other processes when the receiver process is doing a checkpoint, is restarting or it has migrated to another node (reconfiguration phase). The sender observer will fail to communicate, and it will consult the receiver s protector to find about the state of the receiver. The protector will indicate that the process is checkpointing, restarting or has migrated, and the communication will be retried later or to the new location.

The RADICTable and SpareTable were included inside the job information structure (orte_jmap_t) of the Open MPI library. When the parallel application starts, each protector (ORTE daemon) populates its RADICTable and its SpareTable. The RADICTable and SpareTable are updated (on demand) when a protector notices that a process has restarted in another place.

If the application runs out of spares, the default mechanism of RADIC is used (Figure 6.1b).

6.2.2 Experimental Validation

A fault tolerant architecture, generally introduces some kind of overhead in the system it is protecting. These overheads are generally caused by redundancy in some of its forms. The overheads introduced by RADIC are mostly caused by the uncoordinated checkpoints and the pessimistic log mechanism.

These experimental results will show the impact of node failures and how the performance of the system is degraded because of the loss of the computational capacity if there are no spare nodes available. The experimental evaluation that has been done focuses on showing the benefits of automatic failure detection and recovery by managing transparently spare nodes in order to avoid the impact on the performance of applications when resources are lost.

Applications

The applications that we have used in order to test our automatic spare nodes management are the following:

- Matrix Multiplication: It is a MPI based benchmark. The matrix multiplication application is modeled as a master/worker application, the master sends the data to the workers only at the start, and collects the results when the application finalizes (static work distribution among processes). Each application process is assigned to one core during failure-free executions. The matrix multiplication implemented has few communications (only at the beginning and at the end).

- LU Benchmark. It is a MPI based benchmark which is part of the NAS Parallel Benchmarks (NPB) [3].

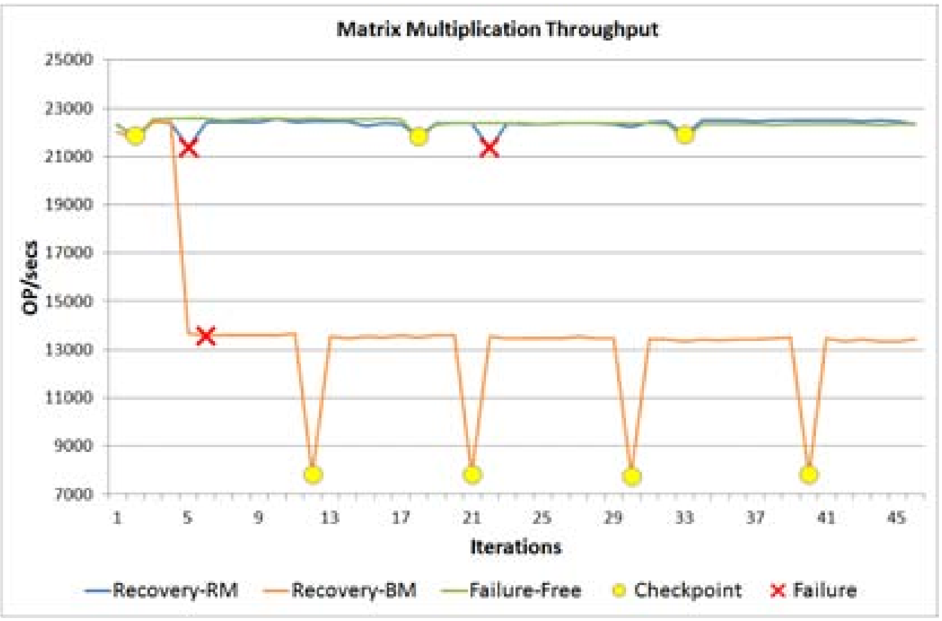
- MPI SMG2000. This is a parallel semi-coarsening multigrid solver that uses a complex communication pattern and performs a large number of non-nearest-neighbor point-to-point communication operations. Application described in detail in [68].

The checkpoint intervals that we use to make the experiments are only for test purposes. If we want to define valid checkpoint intervals we can use the model proposed in [34]. Each value presented as result is a mean of three different independent experiments.

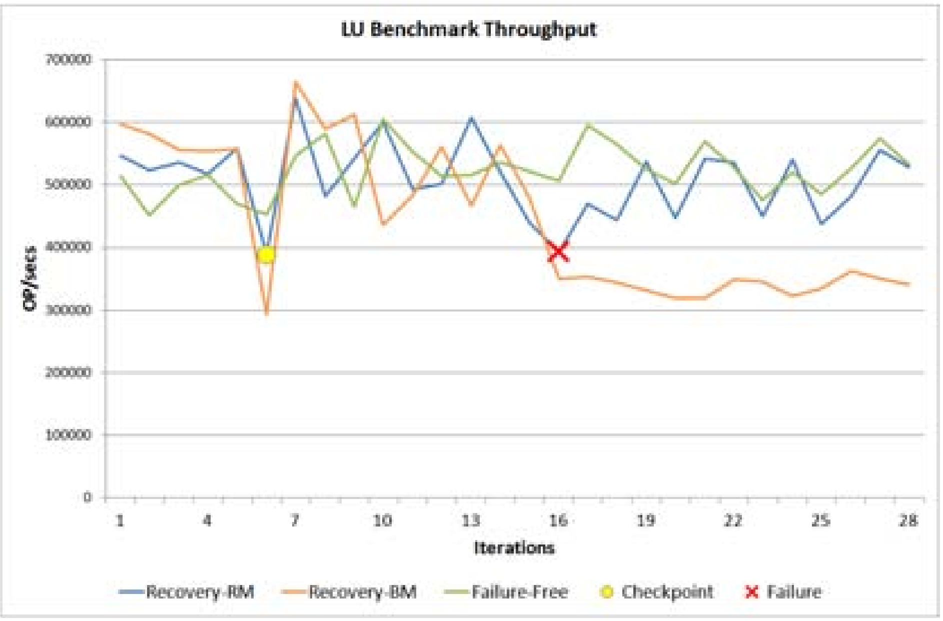
Obtained Results

Our main objective here is to illustrate the application performance degradation avoidance when failures occur in parallel computers. By using spare nodes automatically and transparently to restart failed processes, we can decrease the MTTR of failed processes to a minimum, while maintaining application performance as it was before failure.

As we mentioned before, it is crucial to deal with failures as fast as possible. If the application loses a node and we use the Basic Protection mode of RADIC one of the nodes



(a) Master Worker Matrix Multiplication (32P:8Px4N;2SN). Observing Process 9 in node 2; after restart with the basic protection mode in node 1. After restart with the resilient protection mode process 9 goes to nodes 5 and 6 (Spare Nodes).

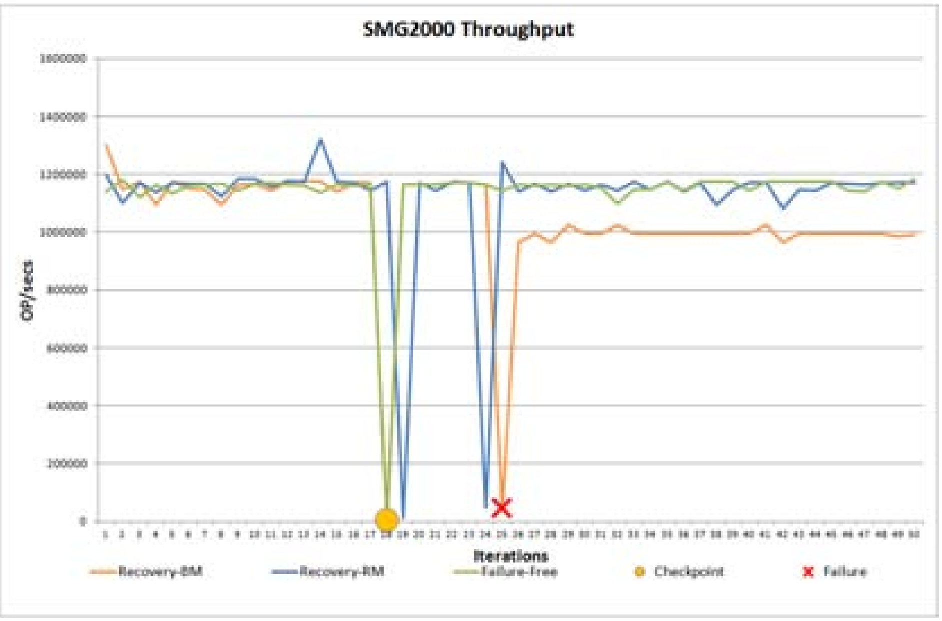


(b) NAS LU C Benchmark (8P:2Px4N;1SN). Observing Process 1 in node 1; after restart with the basic protection mode in node 4. After restart with the resilient protection mode process 1 goes to node 5 (Spare Node).

Figure 6.2: Failure Recovery in RADIC with and without Spare Nodes.

may become overloaded. As a consequence of this, the whole application throughput could decreases.

Replacing failed nodes with spares is not trivial, because it is necessary to include



(c) SMG2000 Application. Observing Process 1 in node 1; after restart with the basic protection mode in node 4. After restart with the resilient protection mode process 1 goes to node 5 (Spare Node).

Figure 6.2: Failure Recovery in RADIC with and without Spare Nodes. (cont.)

the spare node into the parallel environment world and then restart the failed process or processes in it transparently and automatically. Therefore, application performance is affected only by a short period. Following RADIC policies, when restarting a process is not necessary to use a barrier so all non-failed processes know the new location of the restarted process. This operation will be made on demand.

The experiments illustrate how performance (in terms of throughput) is affected after a failure without using spare nodes, and the benefits of using them.

To obtain the operations per second of the Matrix Multiplication application we divided the sub-matrix size that computes each process by the time spent into an internal iteration.

Figure 6.2a illustrates three executions of the Matrix Multiplication benchmark with a matrix size of 3000 and with 50 iterations. For this experiment we have set a test checkpoint interval of 30 secs and we executed the application with 32 processes in 4 nodes with 8 processes per node and 2 spare nodes (32P:8Px4N;2SN). When using the basic protection mode of RADIC (Recovery-BM) and the failure of the node 2 occurs at iteration 5, the application loses around 40% of its initial throughput. However, the application running with the resilient protection mode (Recovery-RM), loses performance only for one iteration due to the restart process and then the initial throughput is maintained. The second failure only is injected in the iteration 21 in the node 5 (which is the spare that replace node 2) of the application with resilient protection.

Figure 6.2b shows the execution of the LU class C benchmark with 8 processes distributed among 4 nodes (8P:2Px4N;1SN). We have only set one checkpoint at 30 secs (iteration 6) and a failure is injected at 75 secs (iteration 16). After the failure and using Recovery-BM the application loses around 35% of throughput, and with Recovery-RM the throughput is reduced in 25% and then is maintained as before the failure. In order to show how throughput is lost, after restart we have used process affinity in the Recovery-BM so restarted process compete with the other application processes residing in the eight-core node.

Finally, in Figure 6.2c we show the behavior of the solver 3 of the SMG2000 benchmark using 8 processes distributed among 4 nodes (8P:2Px4N;1SN), with a size of 300 elements, 3 dimensions and one checkpoint at 100 secs. After the failure at iteration 25 using Recovery-BM, the application loses around 15% of its initial throughput (due to core sharing). However, after the failure at iteration 24 and the restart using Recovery-RM, the application recovers its initial throughput. It is important to mention here that checkpoints and restarts are quite expensive because checkpoints have a size of 2GB each.

As we are aware, the execution time of a faulty application will depend on the moment in which the failure is injected and the checkpoint interval. That is why we focus on show the degradation in terms of throughput when treating faults, and not in the execution time in these experiments.

Considering the obtained results we can conclude that the usage of automatic spare nodes reduce the MTTR of processes since no human intervention is needed. Also spare nodes help to maintain throughput in applications, and of course to avoid system overload and degradation.

6.3 Hybrid Message Pessimistic Logging

In section 4.2, we have described in detail the Hybrid Message Pessimistic Logging (HM_{PL}) and in this section we present the experimental results that show the benefits of this message logging approach. We compare the HM_{PL} with a pessimistic receiver-based message logging in failure-free executions and when applications are affected by failures.

The main goal of the HM_{PL} is to reduce the overhead added in parallel applications during failure-free executions without increasing the time spent in the recovery phase.

In subsection 6.3.1 we are going to present the results obtained during failure-free executions with the HM_{PL} comparing it with the default pessimistic receiver-based message logging (RBML) used in RADIC. Next, in subsection 6.3.2 we are going to compare the recovery times using the RBML and the HM_{PL}. Finally, in subsection 6.3.3 we analyze the overheads obtained in executions with the HM_{PL} and discuss its advantages and limitations.

6.3.1 Comparison of Logging Techniques in Failure-free Executions

In these experiments we measure the impact of each message logging technique without taking into account the impact of checkpoints. The main objective of these experiments is to analyze the impact of two message logging techniques assuming the usage of two different checkpoint strategies: Uncoordinated Checkpoint and Semi-Coordinated Checkpoint [19]. When the strategy used is uncoordinated checkpoint, all message transmissions are logged in a different node. When using a semi-coordinated strategy, only message transmissions that go from one node to another node are saved in a logger residing in another node. The time required to take semi-coordinated checkpoints may be larger than uncoordinated checkpoints since coordination between processes in the same node is needed, but this impact is not analyzed here.

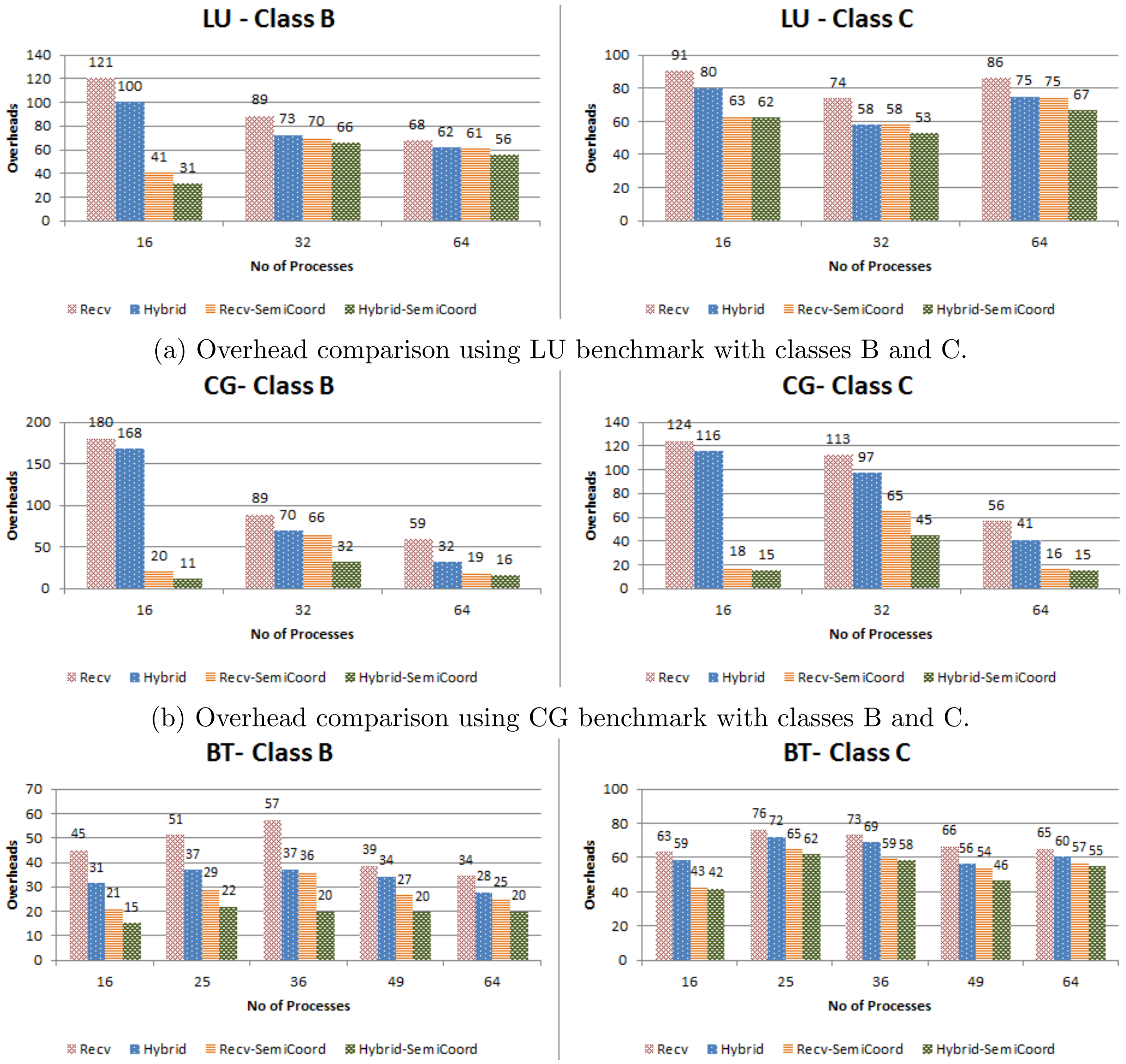
The presented experiments have been made using the RADIC architecture included inside Open MPI. In these experiments, the logger threads share cores with application processes, so there is also an impact in computations but that is homogeneously distributed among processes by using CPU affinity to attach each logger to a core.

As testbed we have used the LU, CG and BT benchmarks from the Nas Parallell Benchmarks (NPB) suit with classes B and C [3]. Each result presented represents a mean of 5 different and independent executions.

In Figure 6.3, we summarize all the results obtained by comparing the Hybrid Message Pessimistic Logging (HM_{PL}) approach with a Pessimistic Receiver-based Message Logging (RBML) approach. Below we explain each type of execution made:

- *Recv:* Executions made using the default pessimistic RBML mechanism of RADIC. All messages are saved in a logger residing in another node, even messages between processes residing in the same node. We are assuming the usage of a fully uncoordinated checkpoint approach.

- *Hybrid:* In these executions we are using the Hybrid Message Pessimistic Logging (HM_{PL}) proposed in this thesis. We are assuming the usage of a fully uncoordinated checkpoint approach.



(a) Overhead comparison using LU benchmark with classes B and C.

(b) Overhead comparison using CG benchmark with classes B and C.

(c) Overhead comparison using BT benchmark with classes B and C.

Figure 6.3: Comparison of overheads using the Hybrid Message Pessimistic Logging and the Pessimistic RBML considering the NAS Parallel Benchmarks.

- *Recv-SemiCoord:* Executions made using the default pessimistic RBML mechanism of RADIC. In this case we are assuming the usage of the Semi-coordinated Checkpoint of RADIC Architecture [19].

- *Hybrid-SemiCoord:* Executions made using the HM_{PL} proposed in this thesis using the Semi-coordinated checkpoint approach of RADIC.

According to the results shown in Figure 6.3, we can observe that for these experiments, the HM_{PL} reduces overheads in all cases. In each column we can observe the overhead introduced by each message logging technique used in comparison to the execution without using message logging. For this set of experiments, we have used a *fill-up* strategy to map processes in each node. Nevertheless, as RADIC requires at least 3 nodes to work properly, experiments with 16 processes have been made with 3 nodes with 7 processes in the first and second node and 2 processes in the third.

Figure 6.3a shows the overheads obtained when executing the LU benchmark. When using the uncoordinated approach of RADIC we can observe an overhead reduction between 6% (Class B, 64 processes) and 21% (Class B, 16 processes). If we use the semi-coordinated checkpoint option of RADIC we can observe an overhead reduction between 1% (Class C, 16 processes) and 10% (Class B, 64 processes).

Figure 6.3b shows the overheads obtained when executing the CG benchmark. When using the uncoordinated approach of RADIC we can observe an overhead reduction of 8% in the worst case scenario (Class C, 16 processes) and 27% (Class B, 64 processes) in the best case. If we use the semi-coordinated checkpoint option of RADIC we can observe an overhead reduction between 1% (Class C, 64 processes) and 34% (Class B, 32 processes).

Results obtained with the BT benchmark are presented in Figure 6.3c. With the HM_{PL} and the uncoordinated checkpoint approach of RADIC we can observe an overhead reduction of 4% in the worst case scenario (Class C, 25 processes) and 20% (Class B, 36 processes) in the best case. If we use the semi-coordinated checkpoint option of RADIC we can observe an overhead reduction between 1% (Class C, 36 processes) and 16% (Class B, 36 processes).

The obtained results show that the HM_{PL} is able to reduce overheads in parallel applications during failure-free executions. However, in the next subsection we will evaluate the recovery times of the HM_{PL} and compare them with a pessimistic RBML.

6.3.2 Experimental Results in Faulty Executions

Here we analyze and compare the impact of the the Hybrid Message Pessimistic Logging (HM_{PL}) approach with a Pessimistic Receiver-based Message Logging (RBML). The main objective of these experiments is to show that the HM_{PL} is able to reduce overheads in applications affected by failures. The Experiments presented here are executed using 3 nodes and 16 processes, the first two nodes with 7 processes and the third one with 2 processes. The uncoordinated checkpoint approach of RADIC is used.

In these experiments we consider that the applications are divided in events, where an event represents the reception of one message in one process. Checkpoints are taken in the same event in each pair of executions. Failures are also injected injected in the same event, in order to properly compare each pair of executions. Failures are injected in the

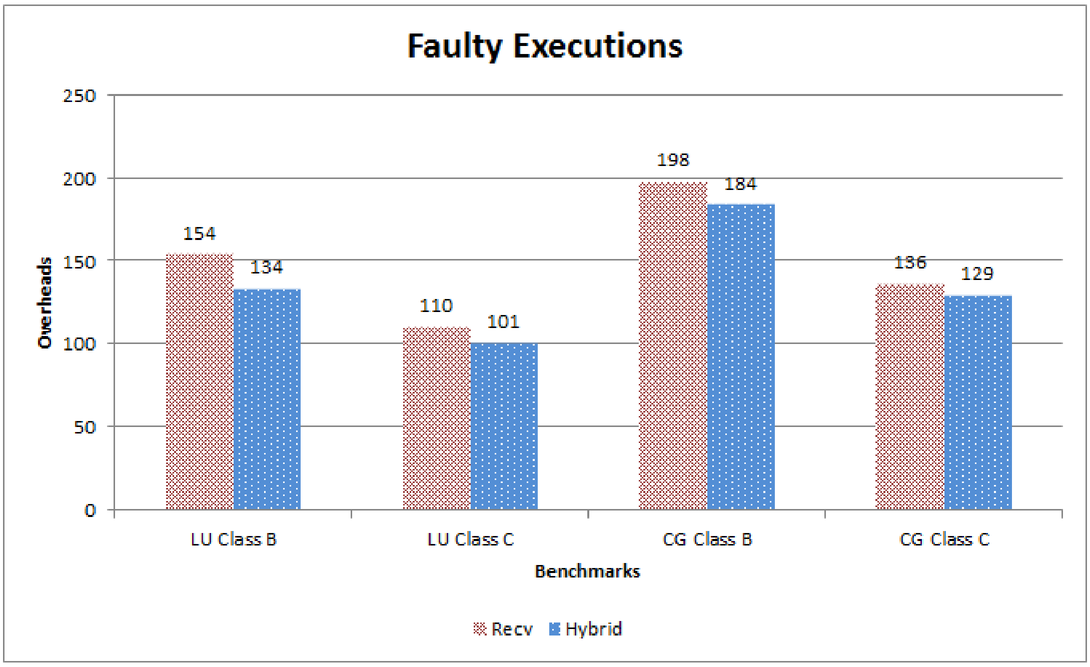


Figure 6.4: Comparison between the Pessimistic Receiver-based Message Logging and the Hybrid Message Pessimistic Logging in executions affected by 1 failure.

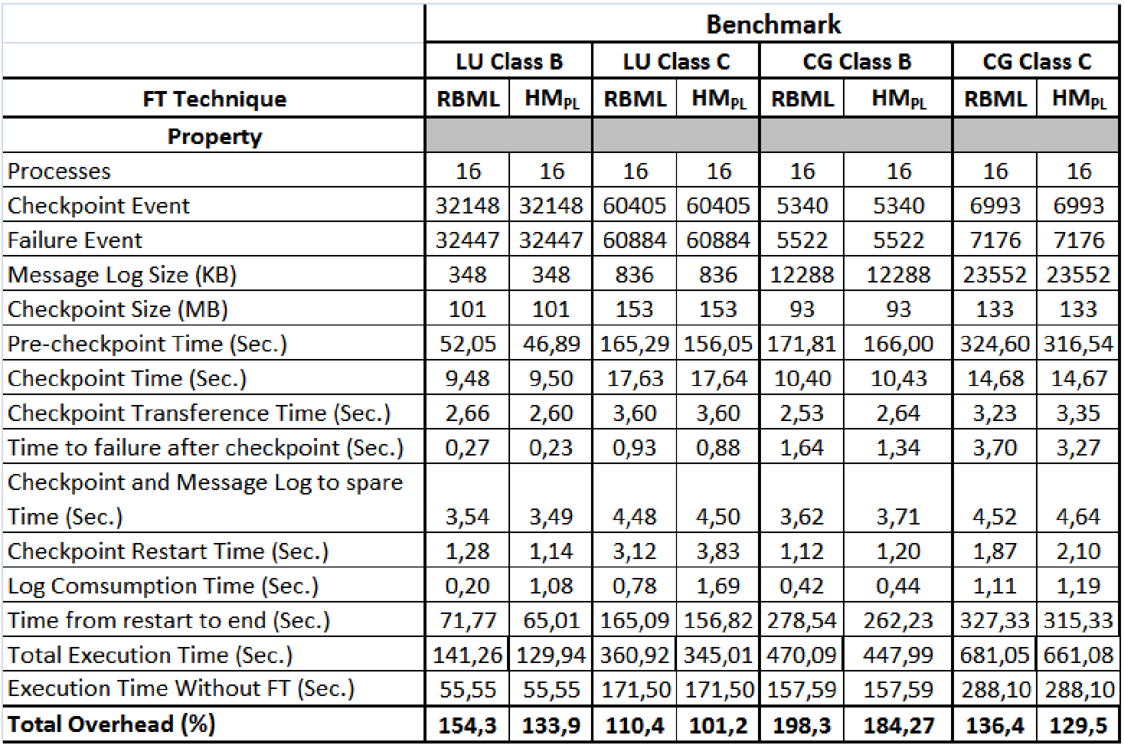
nodes with 2 processes in it. One spare node is used to restart failed processes.

In the executions made, the overheads in computations are homogeneously distributed among processes by using CPU affinity to attach each logger thread to a core.

In Figure 6.4 we summarize the results obtained by comparing the HM_{PL} approach with a Pessimistic RBML approach considering a single failure. This figure summarizes the overheads in faulty executions, using the LU and the CG benchmarks with classes B and C. With the class B of the LU benchmark the HM_{PL} introduces 20% less overhead than the RBML and 9% less overhead with the class C. Considering the class B of the CG benchmark the HM_{PL} introduces 14% less overhead than the RBML, and 7% less considering class C.

In Figure 6.5 is presented a breakdown of the times obtained when injecting a single failure in the parallel executions. In order to extract these measures we have inserted timers inside the RADIC architecture that allow us to determine time consumption of each step of the executions. The message log sizes and checkpoint sizes are calculated when they are transferred to the spare nodes. Each row of the Figure 6.5 is explained below:

- *Pre-checkpoint Time*: represents the time spent from the beginning of the application till the checkpoint takes place.
- *Time to failure after checkpoint*: represents the time elapsed between the check-



	Benchmark							
	LU Class B		LU Class C		CG Class B		CG Class C	
FT Technique	RBML	HM_{PL}	RBML	HM_{PL}	RBML	HM_{PL}	RBML	HM_{PL}
Property								
Processes	16	16	16	16	16	16	16	16
Checkpoint Event	32148	32148	60405	60405	5340	5340	6993	6993
Failure Event	32447	32447	60884	60884	5522	5522	7176	7176
Message Log Size (KB)	348	348	836	836	12288	12288	23552	23552
Checkpoint Size (MB)	101	101	153	153	93	93	133	133
Pre-checkpoint Time (Sec.)	52,05	46,89	165,29	156,05	171,81	166,00	324,60	316,54
Checkpoint Time (Sec.)	9,48	9,50	17,63	17,64	10,40	10,43	14,68	14,67
Checkpoint Transference Time (Sec.)	2,66	2,60	3,60	3,60	2,53	2,64	3,23	3,35
Time to failure after checkpoint (Sec.)	0,27	0,23	0,93	0,88	1,64	1,34	3,70	3,27
Checkpoint and Message Log to spare Time (Sec.)	3,54	3,49	4,48	4,50	3,62	3,71	4,52	4,64
Checkpoint Restart Time (Sec.)	1,28	1,14	3,12	3,83	1,12	1,20	1,87	2,10
Log Comsumption Time (Sec.)	0,20	1,08	0,78	1,69	0,42	0,44	1,11	1,19
Time from restart to end (Sec.)	71,77	65,01	165,09	156,82	278,54	262,23	327,33	315,33
Total Execution Time (Sec.)	141,26	129,94	360,92	345,01	470,09	447,99	681,05	661,08
Execution Time Without FT (Sec.)	55,55	55,55	171,50	171,50	157,59	157,59	288,10	288,10
Total Overhead (%)	**154,3**	**133,9**	**110,4**	**101,2**	**198,3**	**184,27**	**136,4**	**129,5**

Figure 6.5: Breakdown of Recovery Times in executions affected by 1 failure.

point and the failure injection.

- *Checkpoint and Message Log to spare Time*: is the time spent in transferring data to the spare node.
- *Checkpoint Restart Time*: is the time spent in restarting the process from checkpoint.
- *Log Consumption Time*: represents the time spent in reading messages from the message log. When using the HM_{PL}, it also includes the time spent in copying messages from the temporary buffer of senders.
- *Time from restart to end*: is the time elapsed between restart finalization and application ending.

The main difference between the pessimistic RBML and the HM_{PL} can be observed in the *Log Consumption Time* row, since when using the HM_{PL} failed processes should ask some messages to their senders and this causes an almost negligible overhead.

Recovery times of the BT benchmarks are not presented here because the current RADIC implementation is not able to re-execute properly failed processes that belong to communicators created with the MPI_Comm_split command. The inclusion of restarted processes inside a created communicator will be address in the future.

In faulty executions the HM_{PL} maintains almost the same complexity of the RBML (garbage collection is simple) slightly increasing the time to consume the message log. However, this has an almost negligible impact in the parallel execution, thus the time spent in recovery of the RBML and the HM_{PL} are almost the same. Therefore, the HM_{PL} seems to be a suitable replacement to pessimistic RBML, since it reduces the overhead in failure-free executions with a negligible impact in recovery times.

6.3.3 Limitations and Overhead Analysis

According to what has been showed in the previous experiments, the HM_{PL} performs better than the pessimistic RBML. However, the overheads in some situations and for some specific configurations could be high and may discourage the usage of message logging.

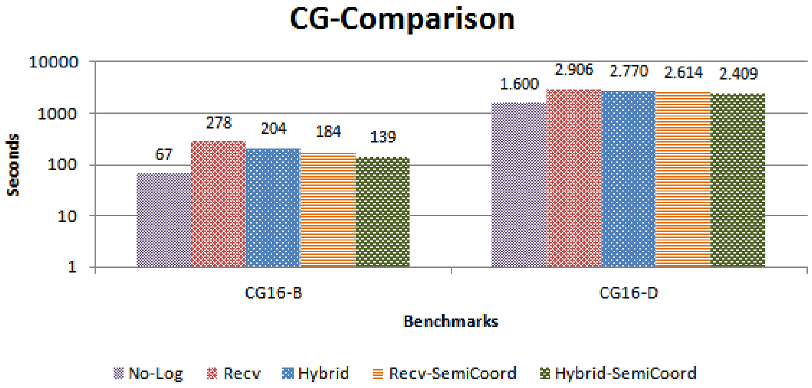
Here we are going to discuss the limitations of the HM_{PL} and analyze possible causes of overheads. Specifically, we will consider the CG benchmark (Figure 6.3b) which presents high overheads in some scenarios. The aim of this discussion is to discover the bottlenecks that avoid the HM_{PL} to perform better and analyze the best case scenarios for this message logging technique.

Figure 6.6 illustrates the execution of the CG benchmark class B and D. In these executions we have put 2 processes per-node, using a total of 8 nodes in order to avoid system overload, so the applications could be analyzed easily. As can be observed, overheads in the class D are lower than in class B. The principal reason of this is that the execution of class B in our execution environment (cluster) is communication-bound.

We have analyzed the execution of each of these benchmarks with the PAS2P tool [85]. We find out that the communications represent 82% of the execution time of the CG class B with 16 processes. Then, any disturbance introduced in the communications will considerably affect the total execution time. Considering the CG class D with 16 processes, the communication time represents 36% of the application, therefore this is a computation bound scenario.

The HM_{PL} focuses on removing blocks from the critical path of the application. However, extra internode messages will be created, and if these transmissions could not be overlapped with computations there is no way to hide overheads. It is important to note that in communication-bound applications, the overheads will be considerable since the HM_{PL} will not be able to overlap the logging step with computation.

In computation bound scenarios, message logging approaches will be able to hide a percentage of the overheads in communications with the computations. Taking into



Overheads (%)		
Logging Types	CG16-B	CG16-D
Recv	316,89	81,64
Hybrid	206,38	73,11
Recv-SemiCoord	175,97	63,40
Hybrid-SemiCoord	108,73	50,55

Figure 6.6: Overhead Analysis of the CG benchmark.

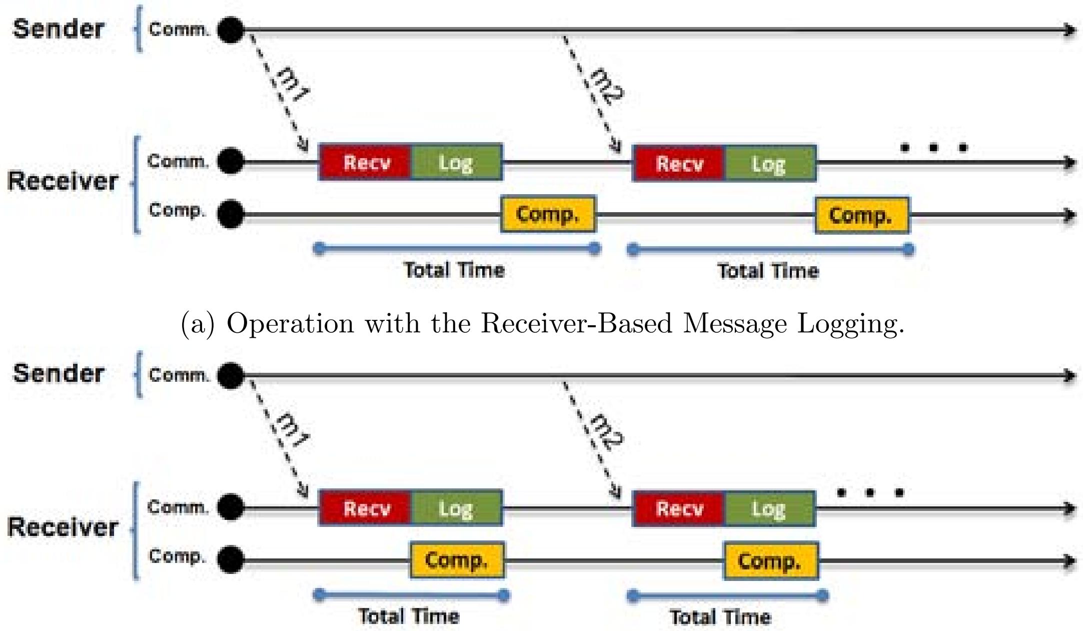
account that the HM_{PL} focuses on removing blocks from the critical path of applications, this gives a better chance to overlap the logging operation with computations.

The HM_{PL} is able to considerably reduce the overheads when comparing it with the RBML as have been seen in subsection 4.3. However, if we consider the computation bound scenario of the CG class D in Figure 6.6 as an example, the HM_{PL} is able to reduce around 13% of overhead in comparison with the RBML. Then, it is important to analyze the possible causes that prevent better results.

We have first analyzed the best scenarios for the HM_{PL}, in order to find out what is the maximum improvement that could be achieved. A synthetic application has been developed in order to determine the maximum potential of the HM_{PL}. In this synthetic application we have a process that sends messages of different sizes to a receiver, and after the reception of each message the receiver do some computation. Figure 6.7a illustrates the behavior of the synthetic application when using a pessimistic RBML technique, and Figure 6.7b shows the execution of the synthetic application using the HM_{PL}. If the computation is enough to cover the transmission of the received message to its protector, then the impact in the execution time should be very low.

Figure 6.8 illustrates the best case scenario for the HM_{PL}, where it is able to reduce up to fourteen times the added overhead of the RBML. In order to achieve such a reduction, receiver processes should execute computation that allows a total overlap of the logging operation.

Scientific parallel applications are normally composed by a set of phases that are



(a) Operation with the Receiver-Based Message Logging.

(b) Operation with the Hybrid Message Pessimistic Logging.

Figure 6.7: Synthetic Application Operation.

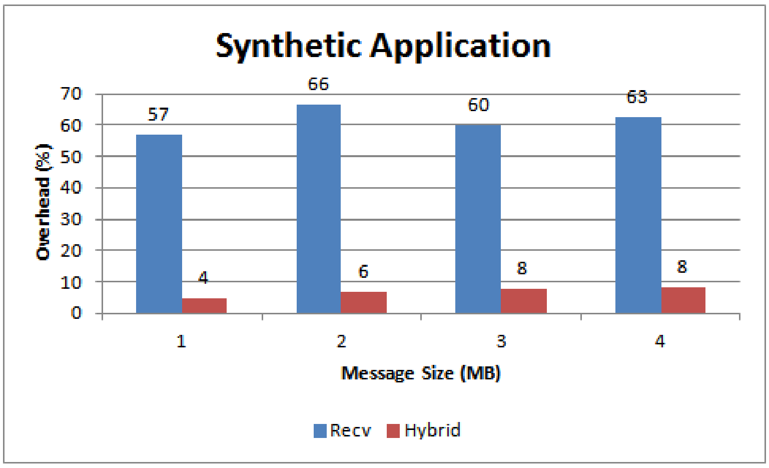


Figure 6.8: Overhead Analysis with the Synthetic Application.

repeated during application execution [85]. Some of these phases could be computation bound and others communication bound. If we use the temporary buffers of the HM_{PL} to save messages during communication bound phases without retransmitting them to the protectors, we can decrease overheads in these phases. Then, in computation bound phases, messages in these temporary buffers could be sent to the loggers threads. This method will allow to reduce impacts in communications, therefore reducing overheads.

In order to apply the above mentioned method, it will be necessary to first analyze and determine application phases. If a tool such as PAS2P is used, we will be able to determine the behavior (communication bound, computation bound, balanced) of each phase and this information could be used by the underlying fault tolerance support to determine the best moment to log messages in the logger threads.

Such a methodology is beyond the scope of this thesis, and thus left as an open line of research.

6.4 Determining suitable Fault Tolerance configurations

In section 5.2, we have discussed the influence in performance that message logging tasks mapping causes. We have proposed to analyze the parallel applications that are going to be executed in order to assign cores for fault tolerance tasks to decrease logging overheads and to save memory. In this section, we present experimental evaluation that has been carried out in order to show that by characterizing parallel applications we are able to find a suitable fault tolerance task mapping.

Most of the overhead added by a receiver-based logging protocol affects communications. In order to lower the impact of a message logging technique we can assign more work per process which allows us to hide the overheads in communications, as it has been discussed in section 5.3. However, if there are no available computational resources for the fault tolerance tasks, the overheads in computations could also become considerable. The main objective of this experimental validation is to prove that by characterizing a parallel application, suitable configurations that reduce the impact of message logging techniques can be found.

When executing a parallel application with fault tolerant support is desirable to store checkpoints and message logs in main memory avoiding the file system, thus allowing fault tolerance mechanisms to execute faster. Also, if we consider an event triggered checkpoint mechanism where checkpoints take place when a message-log-buffer in memory is full and we save memory by executing less application processes per node, we can use a bigger message-log-buffer, thus the checkpoint interval could be longer.

In this section, we focus on determining a suitable message logging task mapping, taking into account the overhead introduced in parallel executions. In order to determine the behavior of parallel applications, tools that characterize and extract traces of the applications are a good option. Such tools could be used when the applications that are going to be executed are complex to analyze manually. In order to select the most suitable message logging task configuration, we propose to use the PAS2P tool [85].

Table 6.1: Process Mapping in an eight-node Cluster.

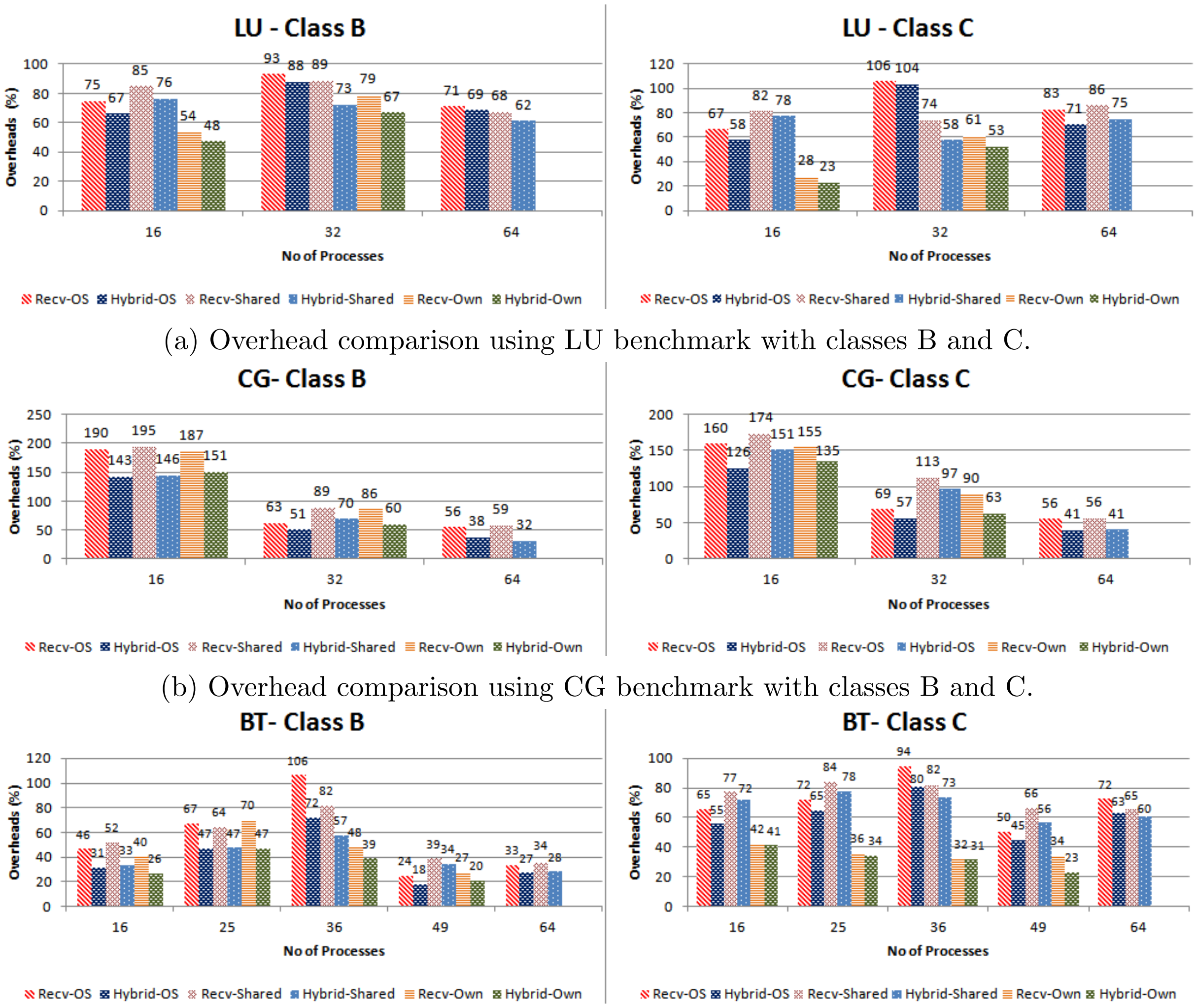
Total	OS-Based & Shared Mapping	Own Resources Mapping
Processes	(Per-node Processes)	(Per-node Processes)
16	3 Nodes: 6 - 6 - 4	3 Nodes: 6 - 6 - 4
25	4 Nodes: 6 - 6 - 6 - 7	4 Nodes: 6 - 6 - 6 - 7
32	4 Nodes: 8 - 8 - 8 - 8	5 Nodes:6 - 6 - 6 - 7 - 7
36	5 Nodes: 7 - 7 - 7 - 7 - 8	6 Nodes: 6 - 6 - 6 - 6 - 6 - 6
49	7 Nodes: 7 - 7 - 7 - 7 - 7 - 7 - 7	7 Nodes: 7 - 7 - 7 - 7 - 7 - 7 - 7
64	8 Nodes: 8 - 8 - 8 - 8 - 8 - 8 - 8 - 8	na

First, we execute the parallel application using the PAS2P tool without fault tolerance in order to extract the parallel signature of the application. Then, the parallel signature is executed with at most three configurations of the message logging tasks. Executing the parallel signature takes 5% of the total execution time in almost all cases. This allows us to quickly determine which strategy introduces less overhead during failure-free executions.

Figure 6.9 and Figure 6.10 show the results obtained assuming an uncoordinated checkpoint approach and a semi-coordinated checkpoint approach. We have used the pessimistic RBML and the HM_{PL} that has been proposed in this thesis as message logging techniques. Results presented here represent means of 3 executions. Three different configurations has been tested:

- OS-Based: No affinity is used to attach logger threads to computational cores. Then, the OS scheduler has freedom to allocate each thread.
- Shared Resources: Affinity is used to attach each logger thread to a computation core, sharing these resources with the application processes.
- Own Resources: Affinity is used to attach logger threads to a subset of cores avoiding the competition for cores between application processes and logger threads.

In order to distribute processes among the available cores, we have mapped the application processes as shown in Table 6.1. As can be seen, there is not one solution that works fine for every configuration. Then, it is crucial to first characterize properly the application in order to determine how we should configure the parallel tasks in combination with the message logging tasks. However, for these experiments the semi-coordinated approach in combination with HM_{PL} are always the right choice taking into account their impact in overheads.



(a) Overhead comparison using LU benchmark with classes B and C.

(b) Overhead comparison using CG benchmark with classes B and C.

(c) Overhead comparison using BT benchmark with classes B and C.

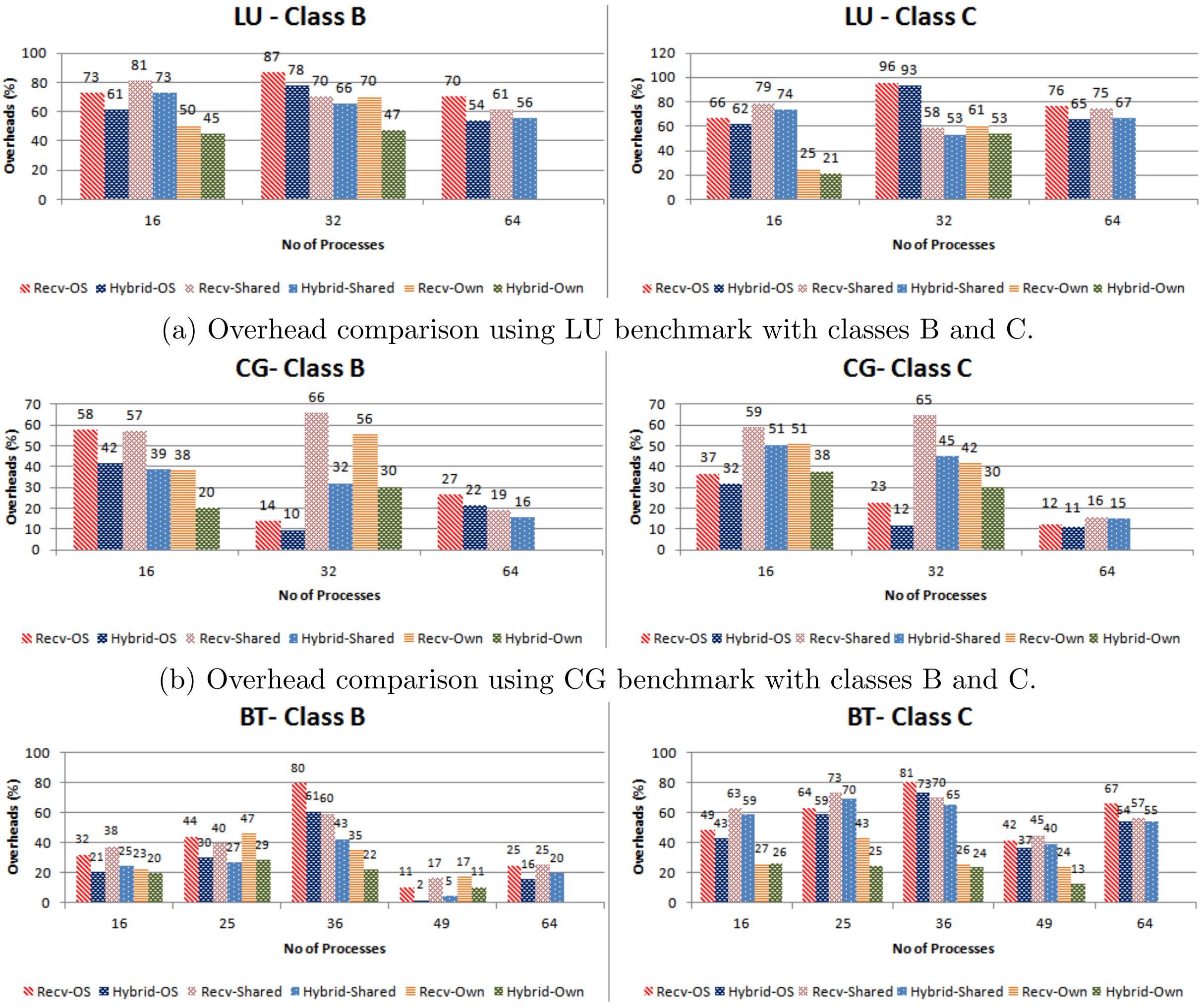
Figure 6.9: Comparison of overheads using the Hybrid Message Pessimistic Logging and the Pessimistic RBML considering the NAS Parallel Benchmarks assuming Uncoordinated Checkpoint.

6.5 Increasing Performability of Parallel Applications

In section 5.3, we have explained in detail our methodology to make efficient executions of SPMD applications using the pessimistic receiver-based message logging (RBML) implemented into RADIC architecture. In this section we show more experiments that validate our analytical model.

6.5.1 Applications

The SPMD applications used to validate our methodology comply with the following characteristics: regular, local and static. Our methodology could only be applied in



(a) Overhead comparison using LU benchmark with classes B and C.

(b) Overhead comparison using CG benchmark with classes B and C.

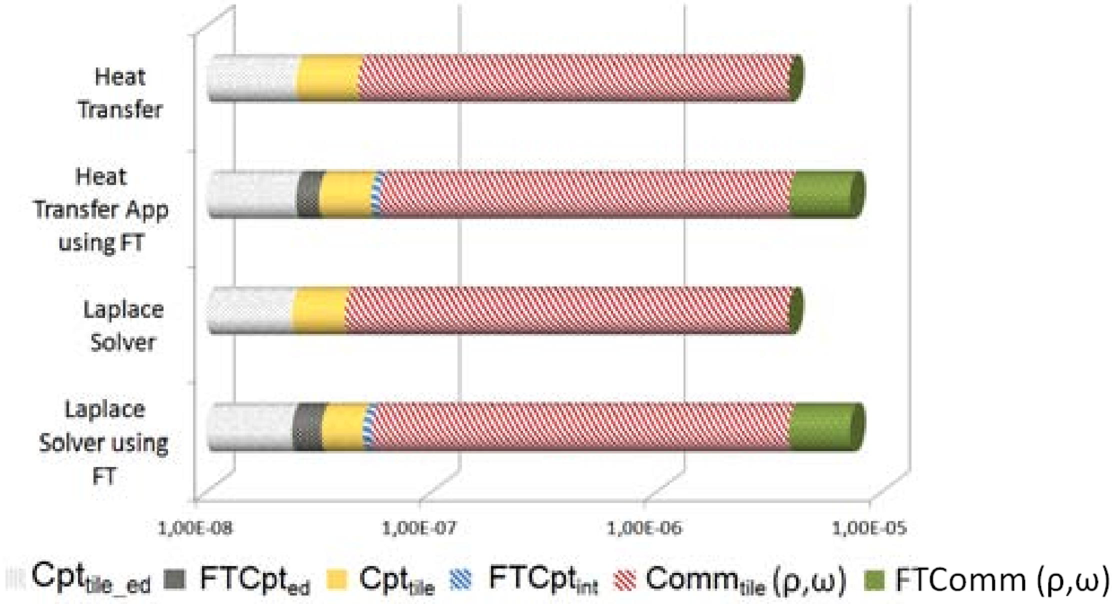
(c) Overhead comparison using BT benchmark with classes B and C.

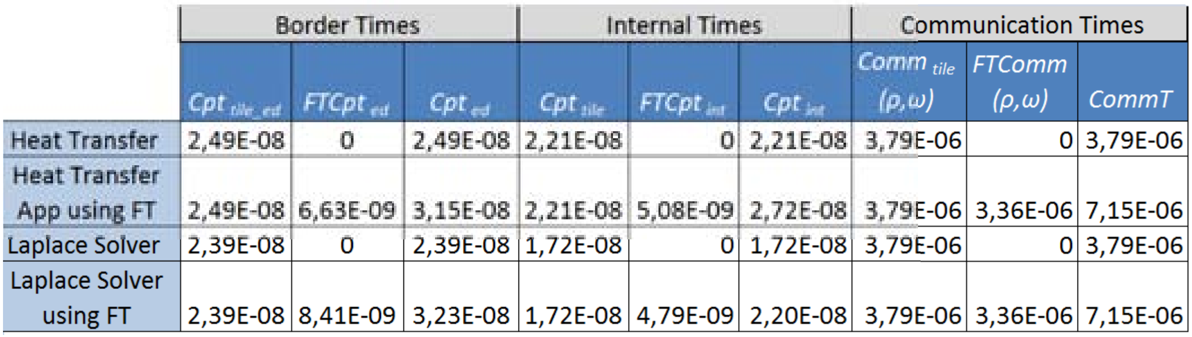
Figure 6.10: Comparison of overheads using the Hybrid Message Pessimistic Logging and the Pessimistic RBML considering the NAS Parallel Benchmarks assuming Semicoordinated Checkpoint.

applications that comply with these characteristics. The applications are:

- Heat Transfer. It follows a Single Program Multiple Data (SPMD) communication pattern, using MPI. This application allows overlapping between the computation steps and the communication steps and it is written using non-blocking communications.

- Laplace Solver. It follows a Single Program Multiple Data (SPMD) communication pattern, using MPI. This application allows overlapping between the computation steps and the communication steps and it is written using non-blocking communications.





	Border Times			Internal Times			Communication Times		
	Cpt_{tile_ed}	$FTCpt_{ed}$	Cpt_{ed}	Cpt_{tile}	$FTCpt_{int}$	Cpt_{int}	$Comm_{tile}$ (ρ,ω)	FTComm (ρ,ω)	CommT
Heat Transfer	2,49E-08	0	2,49E-08	2,21E-08	0	2,21E-08	3,79E-06	0	3,79E-06
Heat Transfer App using FT	2,49E-08	6,63E-09	3,15E-08	2,21E-08	5,08E-09	2,72E-08	3,79E-06	3,36E-06	7,15E-06
Laplace Solver	2,39E-08	0	2,39E-08	1,72E-08	0	1,72E-08	3,79E-06	0	3,79E-06
Laplace Solver using FT	2,39E-08	8,41E-09	3,23E-08	1,72E-08	4,79E-09	2,20E-08	3,79E-06	3,36E-06	7,15E-06

Figure 6.11: Computation and Communication characterization of the Heat Transfer Application and the Laplace Solver for a tile.

In order to carry out the experiments, each application has been executed 5 times, and the results presented here represent the mean of these independent executions.

6.5.2 Prediction Analysis

Firstly, we evaluate the prediction quality achieved with our methodology. The main objective of the experimental validation presented here is to prove that we can determine analytically a suitable number of cores to execute an application, obtaining a maximum speedup taking into account an efficiency threshold while using a message logging technique.

The first phase of our methodology is the *Characterization Phase*. Figure 6.11 illustrates the results obtained for each value necessary to apply our methodology, using two applications with and without message logging. As can be observed, most of the impact of the RBML is concentrated in communications.

Table 6.2: Theoretical and Practical Data of executed Applications

Application	Problem Size	Desired *Effic*(%)	Cpt_{int}	$CommT$	λft	ST	$Ncores$	Obtained *Effic*(%)	abs(Error) (%)
Heat Transfer Tuned	1000x1000	75	$2\ 2E^{-8}$	$3\ 79E^{-6}$	172.09	133.03	56	76	1
Heat Transfer Tuned With FT	1000x1000	75	$2\ 71E^{-8}$	$7\ 15E^{-6}$	263.96	201.95	24	80.4	5.4
Laplace Solver Tuned	1400x1400	85	$1\ 72E^{-8}$	$3\ 79E^{-6}$	220.12	191.07	54	83.57	1.43
Laplace Solver Tuned With FT	1400x1400	85	$2\ 2E^{-8}$	$7\ 15E^{-6}$	325.15	280.36	24	88.94	3.94

Once the necessary data has been extracted in the characterization phase and the desired efficiency value has been selected, Equation 5.14 can be used to obtain the ideal Supertile size (K^n) that allows us to overlap communication with computations. Then, by applying Equation 5.6 the suitable number of nodes can be obtained.

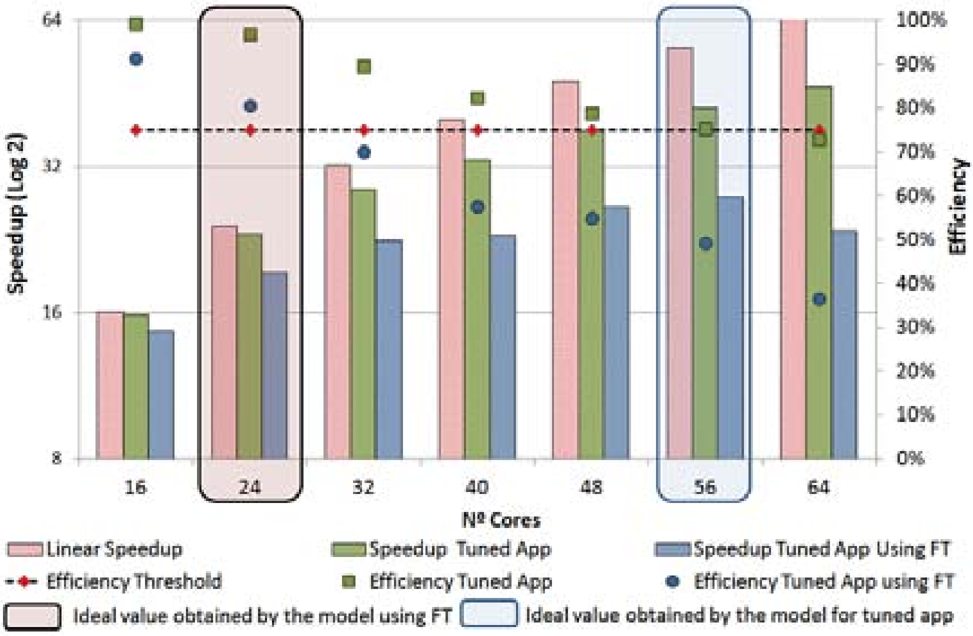
$$Ncores = M^n \quad K^n \tag{5.6}$$

$$K^2 - 4 * K - effic * \lambda ft(\rho \quad) * K + 4 = 0 \tag{5.14}$$

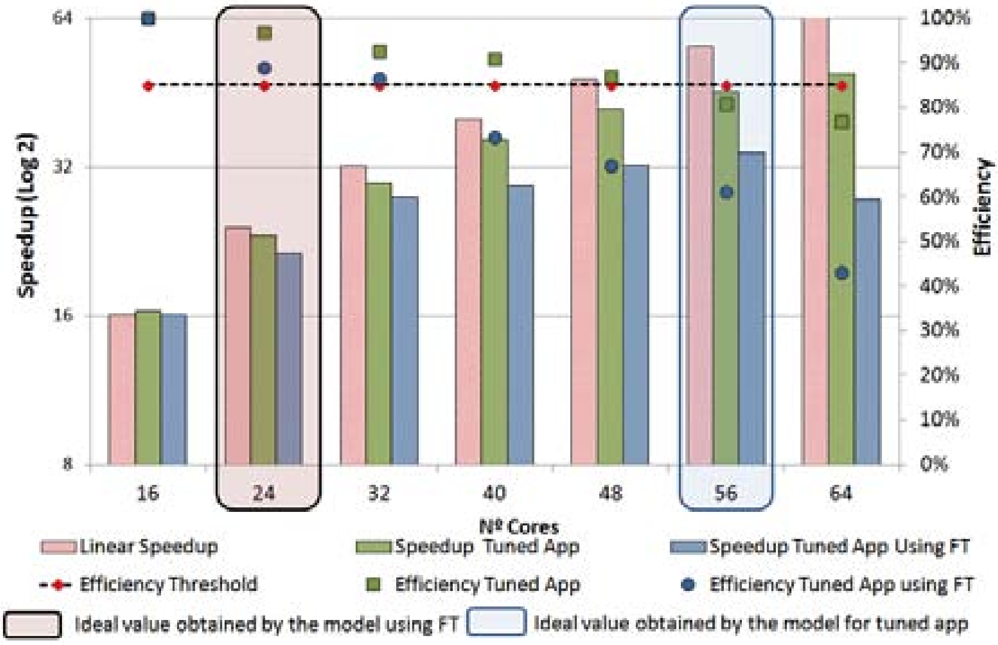
Table 6.2 reveals the data obtained in the characterization phase and the values obtained by applying Equation 5.14 for the two tested applications. The *Obtained Effic(%)* columns presents the real efficiency values obtained when executing each application. When solving the equations, the values *Ncores* would probably have decimal values, the row *Ncores* shows rounded values. It is important to highlight that the mapping and scheduling techniques used are detailed in subsection 5.3.3 and subsection 5.3.4.

In Figure 6.12a we show executions of the Heat Transfer Application, in this example we have fixed the efficiency threshold($Effic$) value at 75%. Considering an execution of a tuned application without using the message logging approach, our model tells us that the highest speedup under the defined efficiency is achieved using 56 computation cores. In order to prove our prediction, we have executed the application using 56 cores obtaining a 1% efficiency error. However, when using the logging protocol the whole scenario changes because of the impact in communication and computation. Figure 6.12a shows us that executing with 56 computation cores, using fault tolerance, will allow us to achieve about 50% efficiency.

Nevertheless, in order maintain our execution near to the 75% threshold of efficiency using the FT approach, we recalculate the values using our model. In this case, it is determined that application with fault tolerance support has to be executed with 24 cores



(a) Heat Transfer App - Size: 1000x1000.



(b) Laplace Equation Solver - Size: 1400x1400.

Figure 6.12: Performance and Prediction Analysis using the Methodology to improve Performability of SPMD Applications.

in order to maintain the relationship between efficiency and speedup. The maximum error rate obtained was 5.4%.

Another example tested was the Laplace solver (Figure 6.12b). In this case we fix our efficiency threshold to 85%, and the ideal number of computation cores to be used without the logging protocol is about 54. However, when the application is going to be executed using the logging protocol, the maximum speedup under our threshold is achieved with 24 computation cores, and when using more cores the efficiency starts to get considerably worse. It is important to notice that we have considered 56 cores in order to maintain the scale of the Figure 6.12b, but for that case the methodology recommends the usage of 54 cores (as shown in Table 6.2).

Figure 6.12a and Figure 6.12b reveal that when scaling up to 64 cores, the speedup of the application with message logging starts to decrease. The message logging protocol would scale side-by-side with applications, but when the application becomes communication bound, the message logging technique would aggravate the situation.

6.5.3 Weak Scalability and Overhead Analysis

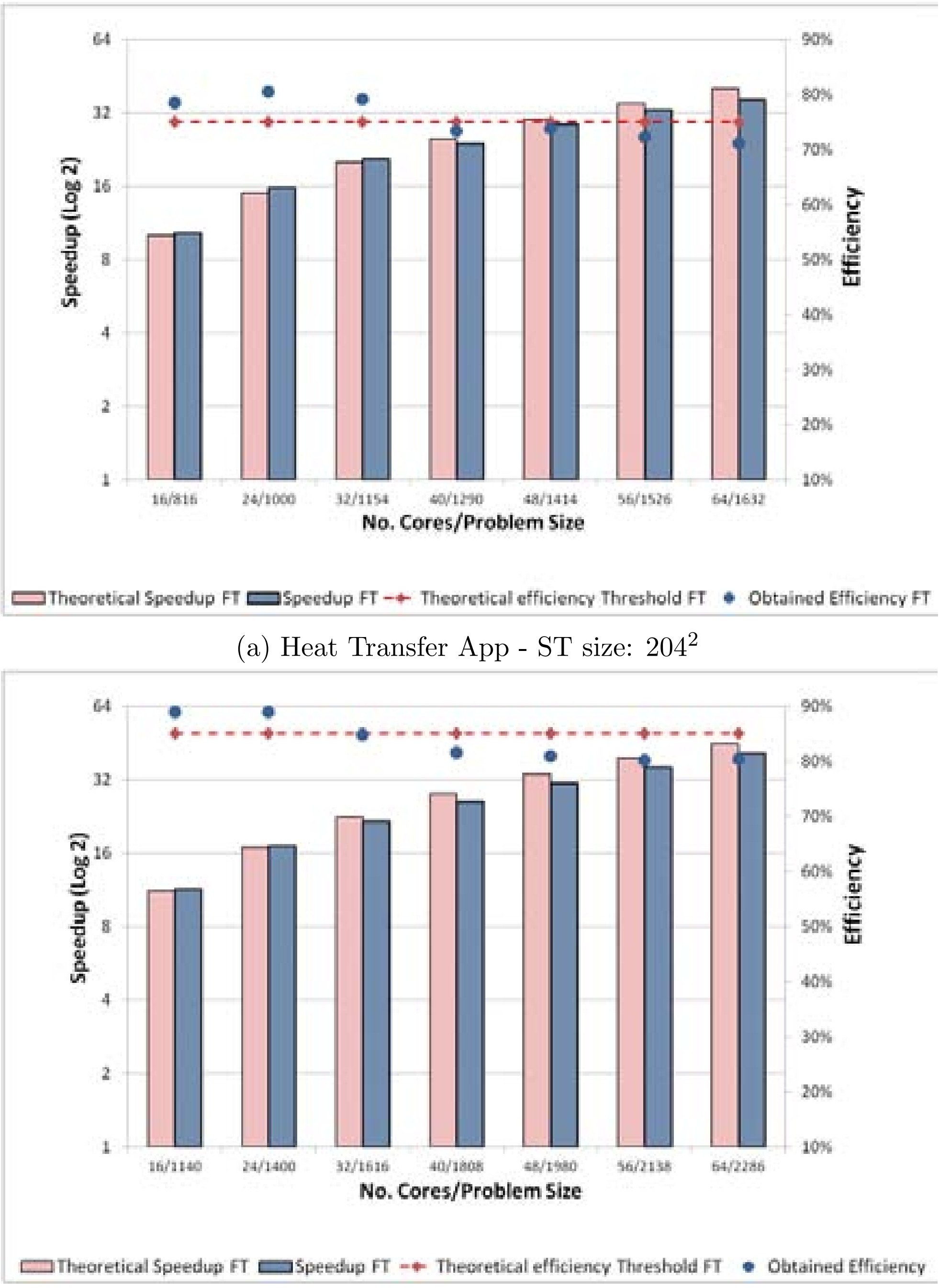
Here we show how the our methodology allows us to determine problem sizes (M) that will scale properly according to the number of cores used ($Ncores$). An analysis of the overhead introduced by the pessimistic RBML approach is also presented here. The objective of this analysis is to prove that when the system is characterized and suitable balance between computation and computation is achieved, the overhead of the message logging technique used is predictable and remains almost constant.

By using the λft values showed in Table 6.2 for experiments with message logging and by applying Equation 5.14, it is possible to determine the Supertile size. Taking into account the Heat Transfer Application, by applying Equation 5.6 we can determine that 24 cores will allow us to obtain near to 75% of efficiency for a problem size of 1000x1000. Then, by fixing $Ncores$ to 24 (without decimal values) and by solving Equation 5.6 for K^n we will obtain a Supertile dimension size of 204. If the same procedure is made with the Laplace Solver with message logging for an efficiency value of 85%, we will obtain a Supertile dimension size of 285.

In Figure 6.13 we show how our methodology allows us to maintain the prediction quality when scaling the size of the problem considering a selected efficiency and maintaining the Supertile size.

In Figure 6.13a the efficiency is fixed to 75% and in Figure 6.13b is fixed to 85%. As can be seen in both cases, the difference between the predicted efficiency value and the obtained efficiency is near 5% for the worst case.

When using the pessimistic RBML, all messages transmissions times are increased and also computational times are affected when processes compete for resources. This was evidenced during the characterization step (Figure 6.11) and it is considered when



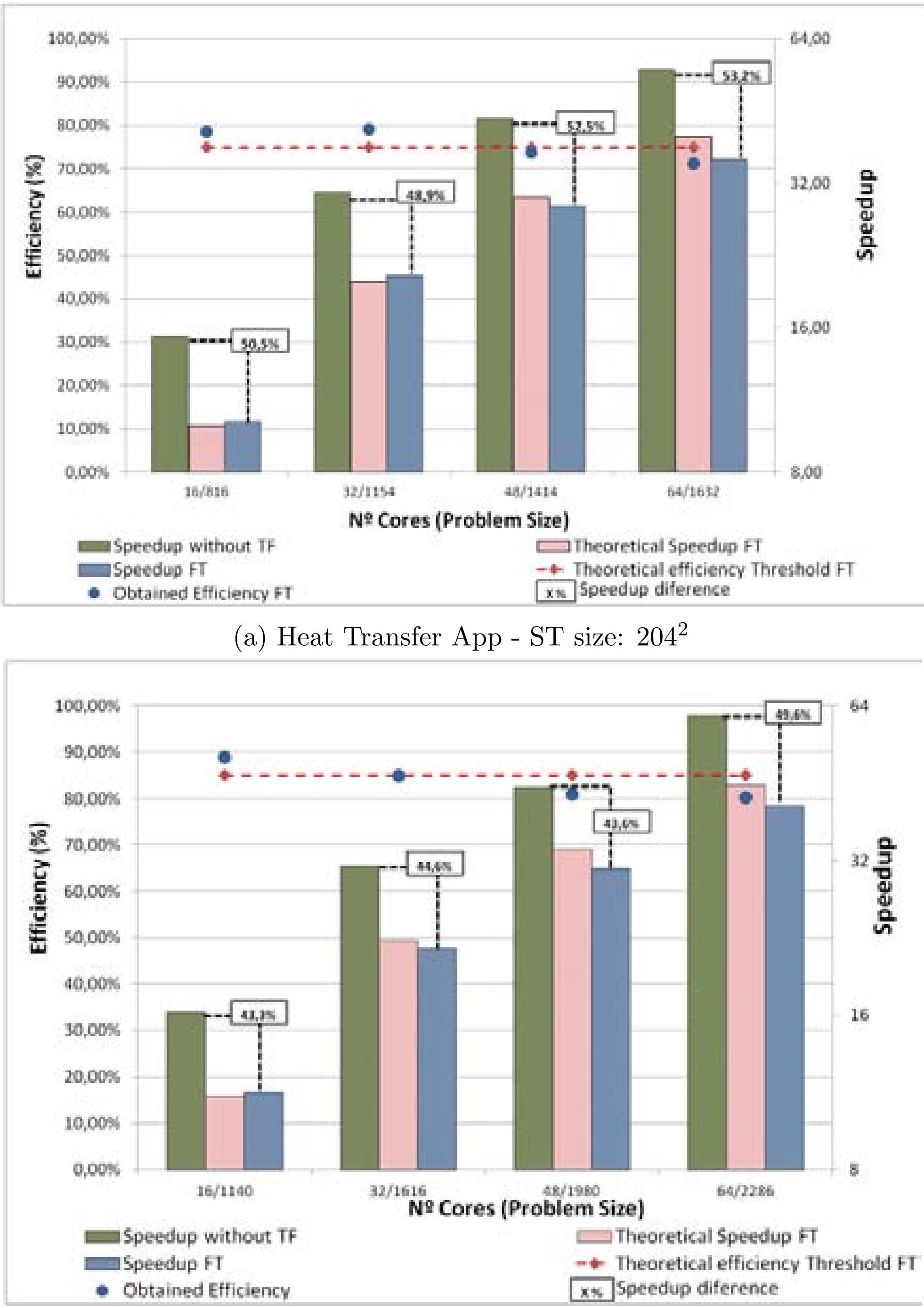
(a) Heat Transfer App - ST size: 204^2

(b) Laplace Equation Solver - ST size: 285^2

Figure 6.13: Weak Scalability Analysis of SPMD applications with Message Logging.

determining the Supertile size as was shown above.

Figure 6.14 shows a comparison in terms of speedup between tuned SPMD applications and tuned SPMD applications using the pessimistic RBML. When executing both applications we properly scale the application workload maintaining the per-process workload.



(a) Heat Transfer App - ST size: 204^2

(b) Laplace Equation Solver - ST size: 285^2

Figure 6.14: Overhead Analysis of SPMD applications with Message Logging.

In Figure 6.14a we show different executions using the Heat Transfer application and we see how when scaling the size of the problem maintaining the Supertile size the overhead introduced remains around 50%. The predicted Speedup and efficiency also have a very small error when comparing them with the real measures.

Figure 6.14b shows the results obtained with the Laplace Equation solver. Again, when scaling the application workload maintaining the relationship between communication and computation the overhead introduced is maintained around 45%.

It is very important to consider that applications are paying these overheads in exchange for protection against node failures.

Experiments have shown that characterizing the message logging effects in computation and communication for SPMD applications, allows us to provide a better number of processes or computational cores to be used in order to improve resource utilization while increasing performability.

Chapter 7

Conclusions

In the previous chapters, we have presented the conception, design, implementation and evaluation of the main contributions of this thesis. The first of these contributions is the Hybrid Message Pessimistic Logging (HM_{PL}). This technique aims to reduce the impact of pessimistic receiver-based message logging during failure-free executions without harming the recovery phase of this protocol.

The second contribution focus on increasing the performability of parallel applications. In order to achieve this, we have proposed and evaluate the advantages of automatic management of spare nodes in HPC clusters to minimize the degradation of applications when failures occur. We have also propose a methodology that allows us to evaluate configurations of message logging tasks of the RADIC architecture, in order to select the one that reduces the disturbance in failure-free executions.

Finally, as third contribution we have presented a methodology that allows to obtain the maximum speedup for a defined level of efficiency when executing SPMD applications with a message logging protocol.

Throughout this thesis, we have discussed the objectives and methods, up to the testing and validation of proposals. Having completed all the stages that comprise our research, we are now able to present the final conclusions and further work of this thesis.

7.1 Final Conclusions

In this thesis, we have defined improvements to current message logging techniques that benefit the execution time of applications that run with a transparent and scalable fault tolerant support. The policies proposed in this thesis aim to reduce the impact of fault tolerant techniques and they have proved to be suitable solutions to improve the performance and dependability of parallel applications.

Hybrid Message Logging Approach

One of the main objectives of this thesis was to reduce to negative impact of message logging techniques in parallel executions. In order to comply with this objective, we have proposed the Hybrid Message Pessimistic Logging. The HM_{PL} is a novel pessimistic message logging approach that combines the advantages of two of the most classical message logging approach: Sender-based Message Logging and Receiver-based Message Logging. The HM_{PL} approach focuses on providing a fault tolerant solution with low MTTR of processes by lowering the complexity of the recovery process of Sender-based approaches and at the same time reducing the impact of failure-free executions in comparison with receiver-based approaches.

This work relies on the usage of data structures to save messages temporarily (in senders and receivers) and allowing the application to continue its execution without restricting message emissions while other messages are being saved in stable storage.

Increasing Performability of Applications

The automatic management of spare nodes allows failed processes to restart faster and avoid the penalties in throughput that could be caused when overloading nodes in a parallel execution. The overloads in multicore clusters could not only affect computational times, but also when several MPI processes compete for memory bandwidth or network access this two could become a bottleneck. The automatic spare node management have been integrated inside the RADIC architecture and the results prove that the computational capacity after a failure is maintained and processes continue with the same throughput level that they had before a failure.

Another main objective of this thesis was to integrate fault tolerance techniques efficiently into parallel systems, considering applications behavior. Taking into account this objective, we have proposed a methodology that consists on characterizing and analyzing possible configurations of a message logging approach in order to find the most suitable according to application behavior. This is done by characterizing the parallel application (or a small kernel of it) obtaining the computation and communication times and the disturbance caused by the logging approaches. Results obtained have proved that by characterizing parallel applications in combination with the message logging tasks allows us to determine configurations that decrease the overheads during failure-free executions.

Case Study: Methodology to increase Performability of SPMD Applications

In order to meet the performability objective when executing parallel applications on computer clusters, we should consider how the fault tolerance tasks affect applications tuning. Taking into account that the message logging tasks are responsible for most of the overhead in uncoordinated approaches, we have proposed a methodology to characterize the overhead that will be added, and then configure the parallel application in order for it to be executed more efficiently.

Our proposal allows us to provide a new feature to applications that have been tuned to be executed efficiently, this feature is resiliency. Using the proposed methodology, users can determine how well an application will run in terms of speedup and efficiency, without doing exhaustive executions in order to obtain the ideal number of processes or computer cores to be used.

Future works should try to extend our methodology to other parallel paradigms and to a larger number of parallel applications. The analysis of how other fault tolerance tasks (such as checkpoints) affect performance of parallel applications in order to propose performability models is also pending.

7.2 Future Work and Open Lines

This thesis has covered several aspects of scalable fault tolerance techniques and performance analysis. However, this research also gives rise to some open lines and future work, which are:

- Extend the analysis of the Hybrid Message Pessimistic Logging to a more widely set of parallel scientific applications.

- Extract application data that could be helpful to determine the best moments to log messages during application executions. By using the Hybrid Message Pessimistic Logging we can save messages in the temporary buffers during communication bound stages of applications and allow the logging operation to continue during computation bound stages.

- Fully include the support for Semi-coordinated checkpoint inside RADIC-OMPI and test our proposals in combination with this technique.

- Analyze the relationship between message sizes and logging overheads in computations, in order to determine the number of cores that should be saved for message logging tasks.

- Extend the methodology for efficient execution of SPMD applications to other parallel paradigms and to a larger number of parallel applications. The analysis of how other fault tolerance approaches, such as checkpoints, affect performance of parallel applications in order to propose performability models is also pending.

- Another open line that we will address is the inclusion of our fault tolerance techniques in cloud environments. For this purpose, a research internship has been made in the University of Innsbruck in order to improve our knowledge in performance analysis and site configurations in cloud.

7.3 List of Publications

The work presented in this thesis has been published in the following papers.

1. **H. Meyer, M. Castro, D. Rexachs and E. Luque. *Propuestas para integrar la arquitectura RADIC de forma transparente*, In CACIC 2011 - XVII Congreso Argentino de Ciencias de la Computación , pp. 347-356, La Plata, Argentina, October 2011.** [58]
   This paper presents two proposals to integrate the RADIC architecture in message passing systems. First is discussed the integration of RADIC inside a specific MPI library and the is presented an analysis of the integration at socket level.

2. **H. Meyer, D. Rexachs and E. Luque. *RADIC: A Fault Tolerant Middleware with Automatic Management of Spare Nodes*, In PDPTA 2012 - The 2012 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 17-23, Las Vegas, USA, July 2012.** [59]
   This paper proposes the integration and automatic management of Spare nodes inside the RADIC architecture. This proposal allows to reduce the degradation in case of failure by automatically replace failed nodes maintaining the computational capacity.

3. **H. Meyer, R. Muresano, D. Rexachs and E. Luque. *Tuning SPMD Applications in order to increase Performability*, In ISPA 2013 - The**

11th IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 1170-1178, Melbourne, Australia, July 2013. [60]

This paper proposes a methodology to make efficient executions of SPMD applications in multicore clusters with fault tolerant support. This methodology allows to obtain the maximum speedup for a defined level of efficiency while a message logging technique is being used.

4. **H. Meyer, R. Muresano, D. Rexachs and E. Luque.** ***A Framework to write Performability-aware SPMD Applications*, In PDPTA 2013 - The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 350-356, Las Vegas, USA, July 2013.** [62]

   This paper presents a framework that allows application programmers to write SPMD applications that make an efficient resource utilization with fault tolerance support. This work proposes to take into account the impact of a message logging approach while developing applications in order to configure applications executions.

5. **H. Meyer, D. Rexachs and E. Luque.** ***Managing Receiver-based Message Logging Overheads in Parallel Applications*, In CACIC 2013 - XVII Congreso Argentino de Ciencias de la Computación , pp. 204-213, Mar del Plata, Argentina, October 2013.** [61]

   This paper proposes a methodology that allows to properly select the configuration of message logging tasks in order to minimize their impact on parallel applications during failure-free executions. This methodology relies on the characterization of the impact of message logging tasks in communication and computation steps of parallel applications in order to select the most appropriate configuration taking into account applications behavior. This paper has been selected to be published in the book of best papers of the CACIC 2013.

6. **H. Meyer, D. Rexachs and E. Luque.** ***Hybrid Message Logging. Combining advantages of Sender-based and Receiver-based approaches*, In ICCS 2014 - International Conference on Computational Science 2014, To Appear, Cairns, Australia, June 2014.** [63]

   This paper presents the Hybrid Message Pessimistic Logging which focus on lowering the overheads of pessimistic receiver-based message logging techniques while maintaining low recovery times. This work is based on the combination of sender-based and receiver-based message logging in order to remove the blocking behavior of pessimistic approaches by using temporary buffers in senders and receivers to save messages.

Bibliography

[1] A. Agbaria and R. Friedman. Starfish: fault-tolerant dynamic mpi programs on clusters of workstations. In *High Performance Distributed Computing, 1999. Proceedings. The Eighth International Symposium on*, pages 167 176, 1999. doi:10.1109/HPDC.1999.805295.

[2] J. Ansel, K. Aryay, and G. Coopermany. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1 12, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3751-1. URL http://portal.acm.org/citation.cfm?id=1586640.1587579.

[3] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, and L. Dagum. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 1994.

[4] R. Batchu, Y. Dandass, A. Skjellum, and M. Beddhu. Mpi/ft: A model-based approach to low-overhead fault tolerant message-passing middleware. *Cluster Computing*, 7(4): 303 315, 10/01 2004.

[5] M. Beck, J. J. Dongarra, G. E. Fagg, G. A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulous, S. L. Scott, and V. Sunderam. Harness: a next generation distributed virtual machine. *Future Generation Computer Systems*, 15(5-6):571 582, 10 1999.

[6] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. Dongarra. Post-failure recovery of mpi communication capability: Design and rationale. *International Journal of High Performance Computing Applications*, 27(3):244 254, August 01 2013.

[7] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v:

Toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29 38, 2002. doi:10.1109/SC.2002.10048.

[8] A. Bouteiller and T. Hérault. MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging. *Supercomputing, 2003 ACM/IEEE Conference*, 2003.

[9] A. Bouteiller, T. Hérault, G. Krawezik, P. Lemarinier, and F. Cappello. MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI. *IJHPCA*, 2006.

[10] A. Bouteiller, G. Bosilca, and J. Dongarra. Retrospect: Deterministic replay of mpi applications for interactive distributed debugging. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 4757:297 306, 2007. doi:10.1007/978-3-540-75416-9_41. URL http://dx.doi.org/10.1007/978-3-540-75416-9_41.

[11] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra. Reasons for a Pessimistic or Optimistic Message Logging Protocol in MPI Uncoordinated Failure Recovery. In *IEEE International Conference on Cluster Computing (Cluster 2009)*, pages 229 236, New Orleans, États-Unis, 2009.

[12] A. Bouteiller, G. Bosilca, and J. Dongarra. Redesigning the message logging model for high performance. *Concurr. Comput. : Pract. Exper.*, pages 2196 2211, 2010.

[13] A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra. Correlated set coordination in fault tolerant message logging protocols for many-core clusters. *Concurrency and Computation: Practice and Experience*, 25(4):572 585, 2013. ISSN 1532-0634. doi:10.1002/cpe.2859. URL http://dx.doi.org/10.1002/cpe.2859.

[14] B. Bouteiller, P. Lemarinier, K. Krawezik, and F. Capello. Coordinated checkpoint versus message log for fault tolerant mpi. *Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on*, pages 242 250, Dec 2003.

[15] F. Broquedis, N. Furmento, B. Goglin, R. Namyst, and P.-A. Wacrenier. Dynamic task and data placement over numa architectures: An openmp runtime perspective. *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, pages 79 92, 2009. doi:10.1007/978-3-642-02303-3_7. URL http://dx.doi.org/10.1007/978-3-642-02303-3_7.

[16] D. Buntinas. Scalable distributed consensus to support mpi fault tolerance. *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message*

Passing Interface, pages 325 328, 2011. URL http://dl.acm.org/citation.cfm?id=2042476.2042515.

[17] R. Buyya. *High Performance Cluster Computing: Programming and Applications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1999. ISBN 0130137855.

[18] F. Cappello. Fault tolerance in petascale/ exascale systems: Current knowledge, challenges and research opportunities. *Int. J. High Perform. Comput. Appl.*, 23 (3):212 226, Aug. 2009. ISSN 1094-3420. doi:10.1177/1094342009106189. URL http://dx.doi.org/10.1177/1094342009106189.

[19] M. Castro, D. Rexachs, and E. Luque. Adding semi-coordinated checkpoint to radic in multicore clusters. In *PDPTA 2013*, pages 545 551, 2013.

[20] L. Chai, Q. Gao, and D. K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. *Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 471 478, 2007. doi:10.1109/CCGRID.2007.119. URL http://dx.doi.org/10.1109/CCGRID.2007.119.

[21] S. Chakravorty and L. Kale. A fault tolerant protocol for massively parallel systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, pages 212 221, April 2004. doi:10.1109/IPDPS.2004.1303244.

[22] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63 75, Feb. 1985. ISSN 0734-2071. doi:10.1145/214451.214456. URL http://doi.acm.org/10.1145/214451.214456.

[23] J. T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303 312, Feb. 2006. ISSN 0167-739X. doi:10.1016/j.future.2004.11.016. URL http://dx.doi.org/10.1016/j.future.2004.11.016.

[24] M. Domeika. *Software Development for Embedded Multi-core Systems: A Practical Guide Using Embedded Intel Architecture*. Elsevier science and technology, 2008. ISBN 9786611371159. URL http://opac.inria.fr/record=b1130740.

[25] A. Duarte, D. Rexachs, and E. Luque. Increasing the cluster availability using radic. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1 8, Sept 2006. doi:10.1109/CLUSTR.2006.311872.

[26] A. Duarte, D. Rexachs, and E. Luque. An intelligent management of fault tolerance in cluster using radicmpi. In B. Mohr, J. Traff, J. Worringen, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 4192 of *Lecture Notes in Computer Science*, pages 150 157. Springer Berlin Heidelberg, 2006. ISBN 978-3-540-39110-4. doi:10.1007/11846802_26. URL http://dx.doi.org/10.1007/11846802_26.

[27] I. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302 1326, 2013. ISSN 0920-8542. doi:10.1007/s11227-013-0884-0. URL http://dx.doi.org/10.1007/s11227-013-0884-0.

[28] V. Eijkhout. *Introduction to High Performance Scientific Computing.* lulu.com, 2011. ISBN 978-1-257-99254-6.

[29] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375 408, Sept. 2002. ISSN 0360-0300. doi:10.1145/568522.568525. URL http://doi.acm.org/10.1145/568522.568525.

[30] I. Eusgeld and F. Freiling. Introduction to dependability metrics. In I. Eusgeld, F. Freiling, and R. Reussner, editors, *Dependability Metrics*, volume 4909 of *Lecture Notes in Computer Science*, pages 1 4. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-68946-1. doi:10.1007/978-3-540-68947-8_1. URL http://dx.doi.org/10.1007/978-3-540-68947-8_1.

[31] G. Fagg and J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 1908:346 353, 2000.

[32] K. Ferreira, J. Stearley, J. H. Laros, III, R. Oldfield, K. Pedretti, R. Brightwell, R. Riesen, P. G. Bridges, and D. Arnold. Evaluating the viability of process replication reliability for exascale systems. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 44:1 44:12, 2011. doi:10.1145/2063384.2063443. URL http://doi.acm.org/10.1145/2063384.2063443.

[33] L. Fialho, G. Santos, A. Duarte, D. Rexachs, and E. Luque. Challenges and Issues of the Integration of RADIC into Open MPI. *European PVM/MPI Users' Group Meeting*, pages 73 83, 2009.

[34] L. Fialho, D. Rexachs, and E. Luque. What is Missing in Current Checkpoint Interval Models? *2012 IEEE 32nd International Conference on Distributed Computing Systems*, 0:322 332, 2011. ISSN 1063-6927. doi:http://doi.ieeecomputersociety.org/10.1109/ICDCS.2011.12.

[35] M. P. I. Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[36] E. Gabriel, G. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. Castain, D. Daniel, R. Graham, and T. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In D. Kranzlmuller, P. Kacsuk, and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 3241 of *Lecture Notes in Computer Science*, pages 97 104. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-23163-9. doi:10.1007/978-3-540-30218-6_19. URL http://dx.doi.org/10.1007/978-3-540-30218-6_19.

[37] Q. Gao, W. Huang, M. J. Koop, and D. K. Panda. Group-based coordinated checkpointing for mpi: A case study on infiniband. In *Parallel Processing, 2007. ICPP 2007. International Conference on*, pages 47 47, 2007. ISBN 0190-3918. ID: 1.

[38] A. Gómez, L. M. Carril, R. Valin, J. C. Mourino, and C. Cotelo. Fault-tolerant virtual cluster experiments on federated sites using bonfire. *Future Gener. Comput. Syst.*, 34:17 25, May 2014. ISSN 0167-739X. doi:10.1016/j.future.2013.12.027. URL http://dx.doi.org/10.1016/j.future.2013.12.027.

[39] L. A. B. Gomez, N. Maruyama, F. Cappello, and S. Matsuoka. Distributed diskless checkpoint for large scale systems. *Cluster Computing and the Grid, IEEE International Symposium on*, 0:63 72, 2010. doi:http://doi.ieeecomputersociety.org/10.1109/CCGRID.2010.40.

[40] T. J. Hacker, F. Romero, and C. D. Carothers. An analysis of clustered failures on large supercomputing systems. *Journal of Parallel and Distributed Computing*, 69(7): 652 665, 2009. ISSN 0743-7315. doi:http://dx.doi.org/10.1016/j.jpdc.2009.03.007. URL http://www.sciencedirect.com/science/article/pii/S0743731509000446.

[41] P. H. Hargrove and J. C. Duell. Berkeley lab checkpoint/restart (blcr) for linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006. URL http://stacks.iop.org/1742-6596/46/i=1/a=067.

[42] R. Hempel. The mpi standard for message passing. 797:247 252, 1994.

[43] J. C. Y. Ho, C.-L. Wang, and F. C. M. Lau. Scalable group-based checkpoint/restart for large-scale message-passing systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 12, 2008. ISBN 1530-2075.

[44] J. Hursey, J. Squyres, T. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. *In Workshop on Dependable Parallel, Distributed and Network-Centric Systems(DPDNS), in conjunction with IPDPS*, 2007.

[45] J. Hursey, R. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. Solt. Run-through stabilization: An mpi proposal for process fault tolerance. *Recent Advances in the Message Passing Interface*, 6960:329 332, 2011. doi:10.1007/978-3-642-24449-0_40. URL http://dx.doi.org/10.1007/978-3-642-24449-0_40.

[46] K. Hwang, J. Dongarra, and G. C. Fox. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011. ISBN 0123858801, 9780123858801.

[47] E. Jeannot and G. Mercier. Near-optimal placement of mpi processes on hierarchical numa architectures. *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II*, pages 199 210, 2010. URL http://dl.acm.org/citation.cfm?id=1885276.1885299.

[48] D. B. Johnson and W. Zwaenepoel. Sender-based message logging. *In Digest of Papers: 17 Annual International Symposium on Fault-Tolerant Computing*, pages 14 19, 1987.

[49] A. Kayi, T. El-Ghazawi, and G. B. Newby. Performance Issues in Emerging Homogeneous Multicore Architectures. *Simulation Modeling Practice and Theory*, 17(9): 1485 1499, 2009.

[50] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc, San Francisco, CA, USA, 2007. ISBN 0120885255, 9780080492681.

[51] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558 565, July 1978. ISSN 0001-0782. doi:10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563.

[52] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello. Improved Message Logging Versus Improved Coordinated Checkpointing for Fault Tolerant MPI. In *Cluster Computing, 2004 IEEE International Conference on*, volume 0, pages 115 124, Los Alamitos, CA, USA, 2004. IEEE Computer Society. ISBN 0-7803-8694-9. doi:http://doi.ieeecomputersociety.org/10.1109/CLUSTR.2004.1392609.

[53] W.-J. Li and J.-J. Tsay. Checkpointing message-passing interface (mpi) parallel programs. *Fault-Tolerant Systems, 1997. Proceedings., Pacific Rim International Symposium on*, pages 147 152, Dec 1997. doi:10.1109/PRFTS.1997.640140.

[54] Y. Li and Z. Lan. Exploit failure prediction for adaptive fault-tolerance in cluster computing. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 8 pp. 538, 2006. ID: 1.

[55] L. M. Liebrock and S. P. Goudy. Methodology for Modeling SPMD Hybrid Parallel Computation. *Concurr. Comput. : Pract. Exper.*, 20(8):1485 1499, June 2008. ISSN 1532-0626.

[56] Y. Luo and D. Manivannan. Hope: A hybrid optimistic checkpointing and selective pessimistic message logging protocol for large scale distributed systems. *Future Generation Computer Systems*, 28(8):1217 1235, 10 2012.

[57] G. Mercier and J. Clet-Ortega. Towards an Efficient Process Placement Policy for MPI Applications in Multicore Environments. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 5759:104 115, 2009. doi:10.1007/978-3-642-03770-2_17.

[58] H. Meyer, M. C. León, D. Rexachs, and E. Luque. Propuestas para integrar la arquitectura radic de forma transparente. *XVII Congreso Argentino de Ciencias de la Computación, Argentina*, pages 347 356, 2011.

[59] H. Meyer, D. Rexachs, and E. Luque. RADIC: A fault tolerant middleware with automatic management of spare nodes. *The 2012 International Conference on Parallel and Distributed Processing Techniques and Applications, July 16-19, Las Vegas, USA*, pages 17 23, 2012.

[60] H. Meyer, R. Muresano, D. Rexachs, and E. Luque. Tuning spmd applications in order to increase performability. *The 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, Melbourne, Australia*, pages 1170 1178, 2013.

[61] H. Meyer, D. Rexachs, and E. Luque. Managing receiver-based message logging overheads in parallel applications. *XIX Congreso Argentino de Ciencias de la Computación. Mar del Plata, Argentina*, pages 204 213, 2013.

[62] H. Meyer, R. M. D. Rexachs, and E. Luque. A framework to write performability-aware spmd applications. *The 2013 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, USA*, pages 350 356, 2013.

[63] H. Meyer, D. Rexachs, and E. Luque. Hybrid Message Logging. Combining advantages of Sender-based and Receiver-based approaches. *International Conference on Computational Science 2014. Cairns, Australia*, To appear, 2014.

[64] J. Meyer. On Evaluating the Performability of Degradable Computing Systems. *Computers, IEEE Transactions on*, C-29(8):720 731, aug. 1980. ISSN 0018-9340. doi:10.1109/TC.1980.1675654.

[65] R. Muresano, D. Rexachs, and E. Luque. Methodology for Efficient Execution of SPMD Applications on Multicore Environments. *10th IEEE/ACM International Conf. on Cluster, Cloud and Grid Computing, CCGrid 2010,Melbourne, Australia*, pages 185 195, 2010.

[66] R. Muresano, D. Rexachs, and E. Luque. Combining Scalability and Efficiency for SPMD Applications on Multicore Clusters. *The 2011 International Conf. on Parallel and Distributed Processing Techniques and Application. PDPTA'11, Las Vegas, USA*, pages 43 49, 2011.

[67] K. Nagaraja, G. M. C. Gama, R. Bianchini, R. P. Martin, W. M. Jr., and T. D. Nguyen. Quantifying the performability of cluster-based services. *IEEE Trans. Parallel Distrib. Syst.*, 16(5):456 467, 2005.

[68] K. Ngiamsoongnirn, E. Juntasaro, V. Juntasaro, and P. Uthayopas. A parallel semi-coarsening multigrid algorithm for solving the reynolds-averaged navier-stokes equations. In *High Performance Computing and Grid in Asia Pacific Region, 2004. Proceedings. Seventh International Conference on*, pages 258 266, July 2004. doi:10.1109/HPCASIA.2004.1324043.

[69] I. Nielsen and C. L. Janssen. Multicore Challenges and Benefits for High Performance Scientific Computing. *Sci. Program.*, pages 277 285, 2008.

[70] I. Nita, A. Rapan, V. Lazarescu, and T. Seceleanu. Efficient threads mapping on multicore architecture. *8th International Conference on Communications (COMM)*, pages 53 56, June 2010.

[71] M. Panshenskov and A. Vakhitov. Adaptive scheduling of parallel computations for spmd tasks. *Proceedings of the 2007 International Conference on Computational Science and Its Applications - Volume Part II*, pages 38 50, 2007. URL http://dl.acm.org/citation.cfm?id=1802954.1802959.

[72] L. Pinto, L. Tomazella, and M. A. R. Dantas. An experimental study on how to build efficient multi-core clusters for high performance computing. *Computational Science and Engineering, 2008. CSE '08. 11th IEEE International Conference on*, pages 33 40, July 2008. doi:10.1109/CSE.2008.63.

[73] S. Rao, L. Alvisi, and H. Vin. The cost of recovery in message logging protocols. *Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on*, pages 10 18, Oct 1998. ISSN 1060-9857. doi:10.1109/RELDIS.1998.740469.

[74] G. Rodríguez, M. J. Martín, P. González, J. Tourino, and R. Doallo. Cppc: a compiler-assisted tool for portable checkpointing of message-passing applications. *Concurrency and Computation: Practice and Experience*, 22(6):749 766, 2010. ISSN 1532-0634. doi:10.1002/cpe.1541. URL http://dx.doi.org/10.1002/cpe.1541.

[75] G. Rodríguez, M. J. Martín, P. González, J. Tourino, and R. Doallo. Compiler-assisted checkpointing of parallel codes: The cetus and llvm experience. *International Journal of Parallel Programming*, 41(6):782 805, 2013. ISSN 0885-7458. doi:10.1007/s10766-012-0231-8. URL http://dx.doi.org/10.1007/s10766-012-0231-8.

[76] T. Ropars, A. Guermouche, B. Uçar, E. Meneses, L. V. Kalé, and F. Cappello. On the use of cluster-based partial message logging to improve fault tolerance for mpi hpc applications. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part I*, Euro-Par 11, pages 567 578, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23399-9.

[77] S. Sankaran, J. M. Squyres, B. Barrett, V. Sahay, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The lam/mpi checkpoint/restart framework: System-initiated checkpointing. *International Journal of High Performance Computing Applications*, 19(4):479 493, November 01 2005.

[78] G. Santos, L. Fialho, D. Rexachs, and E. Luque. Increasing the availability provided by RADIC with low overhead. In *CLUSTER*, pages 1 8. IEEE, 2009. ISBN 978-1-4244-5012-1. URL http://dblp.uni-trier.de/db/conf/cluster/cluster2009.html#SantosFRL09.

[79] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(4): 337 350, 2010. ISSN 1545-5971. doi:10.1109/TDSC.2009.4.

[80] H. Sekhar Paul, A. Gupta, and A. Sharma. Finding a suitable checkpoint and recovery protocol for a distributed application. In *J. Parallel Distrib. Comput.*, pages 732 749, Orlando, FL, USA, May 2006. Academic Press, Inc. doi:10.1016/j.jpdc.2005.12.008. URL http://dx.doi.org/10.1016/j.jpdc.2005.12.008.

[81] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. NetPIPE: A Network Protocol Independent Performance Evaluator. *In IASTED International Conference on Intelligent Information Management and Systems*, 1996.

[82] L. G. Valiant. A bridging model for multi-core computing. *Proceedings of the 16th Annual European Symposium on Algorithms*, pages 13 28, 2008. doi:10.1007/978-3-540-87744-8_2. URL http://dx.doi.org/10.1007/978-3-540-87744-8_2.

[83] K. N. Vikram and V. Vasudevan. Mapping Data-parallel Tasks onto Partially Reconfigurable Hybrid Processor Architectures. *IEEE Trans. Very Large Scale Integr. Syst.*, 14(9):1010 1023, 2006.

[84] J. B. Weissman. Prophet: automated scheduling of spmd programs in workstation networks. *Concurrency - Practice and Experience*, 11(6):301 321, 1999.

[85] A. Wong, D. Rexachs, and E. Luque. Pas2p tool, parallel application signature for performance prediction. *PARA (1)*, pages 293 302, 2010.

[86] J. Xu, R. Netzer, and M. Mackey. Sender-based message logging for reducing rollback propagation. *Parallel and Distributed Processing, 1995. Proceedings. Seventh IEEE Symposium on*, pages 602 609, Oct 1995. ISSN 1063-6374. doi:10.1109/SPDP.1995.530738.