Universitat Autònoma de Barcelona

Escola d'Enginyeria Departament
d'Arquitectura de Computadors
i Sistemes Operatius

Fault Tolerance Configuration for Uncoordinated Checkpoints

Thesis submitted by **Leonardo Fialho** in partial fulfilment of the requirements for the degree of Philosophiæ Doctor per the Universitat Autònoma de Barcelona. This work has been advised by Dr. **Dolores Isabel Rexachs del Rosario**

Bellaterra, July 2011

Fault Tolerance Configuration for Uncoordinated Checkpoints

Thesis submitted by **Leonardo Fialho** in partial fulfilment of the requirements for the degree of Philosophiæ Doctor per the Universitat Autònoma de Barcelona. This work has been advised by Dr. **Dolores Isabel Rexachs del Rosario**

Bellaterra, July 2011

Dr. Dolores Isabel Rexachs del Rosario
Thesis Advisor

Abstract

Parallel computers are growing in complexity and in number of components. The components miniaturisation and concentration are the major root causes of the failures increasingly seen on these computers. Thus, in order to achieve the execution end, parallel application should use a fault tolerance strategy.

A widely used strategy is the rollback-recovery, which consists of saving the application state periodically. In the event of a fault occurring, the application resumes it execution from the most recent saved state. These fault tolerance protocols include an overhead on the parallel application execution.

Using a coordinated checkpointing protocol it becomes easy to estimate the application execution time, as well as to calculate the frequency in which checkpoints should be taken. In fact, there are very precise models to estimate the application execution time and the checkpoint interval nowadays.

However, the use of the coordinated checkpointing may not be the best solution to provide fault tolerance on the next-generation parallel computers. In other words, the current paradigm of fault tolerance for parallel applications is not suitable for the future parallel computer.

Fault tolerance protocols such as uncoordinated checkpointing permits that each process of the parallel application saves its

state independently of other processes. The combination of uncoordinated checkpointing with logging of message-passing events avoids the inconvenience of this sort of protocol, such as the domino effect and orphan messages. This is the emergent paradigm of fault tolerance for scalable parallel applications.

For instance, there is no model suitable to estimate the execution time of a parallel application protected by uncoordinated checkpointing. As well as there is no convenient model to calculate the frequency in which those checkpoints should be taken.

The objective of this thesis is to define suitable models that can be used with each paradigm: the coordinated and the uncoordinated. These models should provide an estimation of the application wall time clock running under each fault tolerance paradigm, as well a methodology to define the value of the variables used to calculate the checkpointing interval.

The main motivation of this work is to provide at the same time the knowledge necessary to face the emergent fault tolerance paradigm and make it suitable to be used by parallel applications users.

Resumo

A atual tendência dos computadores paralelos é crescer em complexidade e número de componentes. A miniaturização e a concentração destes elementos é a principal causa da aparição e aumento do número de falhas nestes computadores. Para permitir a correta execução das aplicações paralelas nestas máquinas é necessário existir mecanismos para tolerar tais falhas.

Uma estratégia largamente utilizada são as técnicas de *rollback-recovery*, que consistem em guardar periodicamente o estado da aplicação e, em caso de falhas, recuperar a aplicação desde o último estado guardado. O uso destes protocolos gera um incremento no tempo de execução das aplicações.

Quando se utiliza protocolos de *checkpoint* coordenados, é fácil estimar o tempo total de execução de uma aplicação, assim como a frequência na qual os *checkpoint* devem ser criados. Atualmente existem modelos precisos para estimar estes tempos.

Entretanto, o uso de protocolos de *checkpoint* coordenados pode não ser a melhor solução para prover tolerância a falhas nos computadores de próxima geração. Em outras palavras, o atual paradigma de tolerância a falhas para computadores paralelos não é adequado para os sistemas futuros.

Os protocolos de tolerância a falhas não coordenados permitem que, cada processo da aplicação paralela, guarde seu estado, independente dos demais processos; a combinação destes com técnicas de *log* de eventos eliminam os inconvenientes destes protocolos, como o efeito dominó e as mensagens órfãs. Esta combinação representa o paradigma emergente de tolerância a falhas para aplicações paralelas.

Atualmente não existem modelos adequados para estimar o tempo de execução de aplicações paralelas que estão sendo protegidas por *checkpoint* não coordenados. Assim como também não existem modelos para calcular a frequência com que os *checkpoints* devem ser criados.

O objetivo desta tese é definir modelos específicos para cada um dos dois paradigmas: coordenado e não coordenado. Os modelos fornecem uma estimação do tempo total de execução das aplicações quando protegidas por qualquer um dos dois paradigmas. Ademais, se propõe uma metodologia para definir o valos das variáveis necessárias para calcular o intervalo de *checkpoint*.

A principal motivação deste trabalho é prover o conhecimento necessário para enfrentar o paradigma emergente de tolerância a falhas e fazê-lo acessível para os usuários de aplicações paralelas.

Resumen

La tendencia general de los computadores paralelos es crecer en complejidad y en número de componentes. La miniaturización y la concentración de dichos elementos es la principal causa de la aparición y aumento de los fallos en estos computadores. Asimismo, para permitir la ejecución correcta de las aplicaciones paralelas, existe la necesidad de proveer soporte y de tolerar fallos en estos entornos.

Una estrategia amplamente utilizada es el *rollback-recovery*, que consiste en guardar periódicamente el estado de la aplicación y, en caso de fallos, reanudar la aplicación desde el último estado guardado. El uso de estos protocolos añade una sobrecarga al tiempo de ejecución de la aplicación.

Con el uso de protocolos de *checkpoints* no coordinados, es fácil estimar el tiempo total de ejecución de una aplicación, así como también la frecuencia en la cual estos *checkpoints* deben ser guardados. Actualmente, existen modelos precisos para estimar estos tiempos.

Sin embargo, el uso de protocolos de *checkpoints* coordinados, puede no ser la mejor solución para proveer tolerancia a fallos en los computadores paralelos de próxima generación. En otras palabras, el actual paradigma de tolerancia a fallos para computadores paralelos, no es adecuado para los futuros sistemas.

Los protocolos de tolerancia a fallos no coordinados permiten que, cada proceso de la aplicación paralela guarde su estado independientemente de los demás procesos; la combinación de estos protocolos con técnicas de *log* de eventos eliminan los inconvenientes de los protocolos no coordinados, como el efecto domino y la aparición de mensajes huérfanos. Esta combinación representa el paradigma emergente de tolerancia a fallos para aplicaciones paralelas escalables.

Actualmente, no hay modelos adecuados para estimar el tiempo de ejecución de aplicaciones paralelas que están siendo protegidas por *checkpoints* no coordinados. Así como tampoco existen modelos para calcular la frecuencia en que dichos *checkpoints* deben ser creados.

El objetivo de esta tesis es, definir los modelos específicos para cada uno de los paradigmas: el coordinado y el no coordinado. Los modelos proveen una estimación del tiempo total de ejecución de las aplicaciones cuando están protegidas por cualquiera de los dos paradigmas. Además, se propone una metodología para definir el valor de las variables necesarias para calcular el intervalo de *checkpoints*.

La principal motivación de este trabajo es proveer el conocimiento necesario para enfrentar el paradigma emergente de tolerancia a fallos y hacerlo asequible para los usuarios de las aplicaciones paralelas.

Resum

La tendència general dels computadors paral·lels és créixer en complexitat i en nombre de components. La miniaturització i la concentració d'aquests elements és la principal causa de l'aparició i augment de les fallades en aquests computadors. Així mateix, per permetre l'execució correcta de les aplicacions paral·leles, existeix la necessitat de proveir suport i de tolerar fallades en aquests entorns.

Una estratègia àmpliament utilitzada és el *rollback-recovery*, que consisteix a guardar periòdicament l'estat de l'aplicació i, en cas de fallades, reprendre l'aplicació des de l'últim estat guardat. L'ús d'aquests protocols afegeix una sobrecàrrega al temps d'execució de l'aplicació.

Amb l'ús de protocols de *checkpoints* no coordinats, és fàcil estimar el temps total d'execució d'una aplicació, així com també la freqüència en la qual aquests *checkpoints* han de ser guardats. Actualment, existeixen models precisos per estimar aquests temps.

No obstant això, l'ús de protocols de *checkpoints* coordinats, pugues no ser la millor solució per proveir tolerància a fallades en els computadors paral·lels de propera generació. En altres paraules, l'actual paradigma de tolerància a fallades per a computadors paral·lels, no és adequat per als futurs sistemes.

Els protocols de tolerància a fallades no coordinades permeten que, cada procés de l'aplicació paral·lela guardi el seu estat independentment dels altres processos; la combinació d'aquests protocols amb tècniques de *log* de missatges eliminen els inconvenients dels protocols no coordinats, com l'efecte domino i l'aparició de missatges orfes. Aquesta combinació representa el paradigma emergent de tolerància a fallades per a aplicacions paral·leles escalables.

Actualment, no hi ha models adequats per estimar el temps d'execució d'aplicacions paral·leles que estan sent protegides per *checkpoints* no coordinats. Així com tampoc existeixen models per calcular la freqüència en què aquests *checkpoints* han de ser creats.

L'objectiu d'aquesta tesi és, definir els models específics per a cadascun dels paradigmes: el coordinat i el no coordinat. Els models proveeixen una estimació del temps total d'execució de les aplicacions quan estan protegides per qualsevol dels dos paradigmes. A més, es proposa una metodologia per definir el valor de les variables necessàries per calcular l'interval de *checkpoints*.

La principal motivació d'aquest treball és proveir el coneixement necessari per enfrontar el paradigma emergent de tolerància a fallades i fer-ho assequible per als usuaris de les aplicacions paral·leles.

Acknowledgements

Four year ago I left my country in order to achieve a Ph.D. Now, in the end of this journey I am finishing it. More than this degree I have found a new life and new friends.

Of course, there are lots of people whom I need to thank. Starting from the principle, thank to my perfect family which has supported me in all my decisions. Terezinha, Eunápio, Erick and Diana. I have no words to describe how perfect you are.

It is amazing to imagine that a guy, whom does not like to study, is crossing this long and painful journey. The responsible for this is Dolores Rexachs, my lovely advisor. Of course, I can not forget Emilio Luque, which for me will be always an example of researcher. This work only exists because I have these two "kindly" and special tutors. Lola and Emilio, you gave me, and are still giving, all "nutrients" necessary to develop my work here.

Thanks too for all other people who are part of the "CAOS" department. Thanks to all professors and, in special, two guys: Daniel Ruiz and Javier Navarro.

At least, I have to thanks to the doctor Miguel Brandão who cured me of a skin cancer. You know, there is no sense in being a Ph.D. if you are sick.

Agradecimentos

Os agradecimentos em português vão apenas para duas pessoas especiais: Eunápio e Terezinha.

Voces sabem que em nossa família não é muito comum expressar palavras “melosas”, então: Obrigado!

To myself...

Contents

List of Figures

List of Tables

List of Equations

I know but one freedom, and that
is the freedom of the mind.
— **Antoine de Saint-Exupéry**

1

Introduction

Parallel computers are growing in complexity and in number of components. At the same time computers components are becoming smaller and faster. The current trend to increase computation power is to concentrate even more components in a single machine and to combine several machines to work together. However, the components miniaturisation and concentration are the major root causes of the failures increasingly seen on these computers.

Assuming that, along the years, parallel computers will fail more frequently it is possible that applications running on these machines never reach completion (46, 78, 79). It is especially true for long-running applications. In order to achieve the execution end, parallel application should use a fault tolerance strategy. For that reason fault tolerance has become an important issue for parallel applications in the last few years.

Notwithstanding the current interest in fault tolerance for parallel applications, this is not a new concern to applications users and researchers. Fault tolerance has being studied from many years, since the 70's it becomes even more important (25). And in the last four decades several strategies have been developed to deal with computer failures.

A widely used strategy is the rollback-recovery, which consists of saving the application state periodically. In the event of a fault occurring,

the application resumes it execution from the most recent saved state. This state-saving technique is simple to deploy in comparison to writing applications using fault tolerant algorithms.

Currently there are well-defined rollback-recovery protocols to handle computer failures in order to avoid losing all the computation time. To deal with failures in parallel computers, these protocols have evolved and other protocols have been proposed also. This was necessary mostly because on parallel computers applications often use the message-passing model to communicate processes.

Actually, many of the rollback-recovery protocols that protect parallel applications are based on checkpointing the application processes or saving the message-passing communication events. As usual, more elaborated strategies tends to be harder to use. However, some fault-tolerance strategies combine both techniques.

This is one of the reasons that explain the popularity of checkpointing. This state-saving technique is really simple to deploy on a single-process application. The same technique has been adapted to perform a checkpoint of a parallel application composed of several processes. The technique of checkpointing all processes of a parallel application is known as coordinated checkpointing.

To perform a checkpointing the application execution should be interrupted. However it is expected that these periodically interruptions introduce less overhead than the time needed to re-execute the entire application.

Using a coordinated checkpointing protocol it becomes easy to estimate the application execution time, as well as to calculate the frequency in which checkpoints should be taken (44, 86).

In fact, there are very precise models to estimate the application execution time and the checkpoint interval nowadays (29). However, these models have been designed to be used with a single-process application.

The coordinated checkpointing behaviour allows the use of the same model for parallel application being protected by a coordinated checkpointing protocol.

The easiness to deploy and the existence of precise models makes the coordinated checkpoint the *de facto* paradigm of fault tolerance for parallel applications. However, will this paradigm still be valid for future computer generations?

There are at least three good reasons to think that the coordinated checkpointing paradigm will not be valid for the future computers generations [1]. All these reasons are based on contrasting the intrinsic characteristics of the coordinated checkpointing with the parallel machines evolution trends:

— Growing the total memory size of the parallel computer *versus* increasing the stable storage throughput (the coordinated checkpoint protocol requires to save the global application state periodically);

— Growing the number of computing nodes *versus* the time needed to coordinate all process in the parallel application (most protocols require at least two global coordinations (34));

— Growing the number of components which form the parallel machine (increasing the fault frequency) *versus* the computation time loss of all processes in case of fault while using coordinated checkpointing protocols.

These statements points that the use of the coordinated checkpointing may not be the best solution to provide fault tolerance on the current parallel computers. In other words, the current paradigm of fault tolerance for parallel applications is not suitable for the future parallel computer generations.

[1] These reasons will be explained and discussed in details in Section 2.

The current scenario of high performance computing claims for solutions that avoid collective operations such as the coordination. This requirement tends to becomes even header for the future parallel computer generations if computers evolve in the same manners as they are actually evolving. Fortunately, these solutions already exist.

Fault tolerance protocols such as uncoordinated checkpointing permits that each process of the parallel application saves its state independently of other processes. And the combination of uncoordinated checkpointing with logging of message-passing events avoids the inconvenience of this sort of protocol. This is the emergent paradigm of fault tolerance for parallel applications.

The emergent paradigm in rollback-recovery fault tolerance for parallel applications becomes to solve the scalability problem of providing fault tolerance for parallel applications. However, there is a lack of knowledge concerning the emergent paradigm, *i.e.* the use of protocols that combine uncoordinated checkpointing with event logging.

For instance, there is no model suitable to estimate the execution time of a parallel application protected by uncoordinated checkpointing. As well as there is no convenient model to calculate the frequency in which those checkpoints should be taken. This frequency should balance the overhead during the fault-free execution and the recovery time in case of fault.

The optimum checkpoint interval defines the frequency in which checkpoint should be taken to introduce the minimum overhead on the application. This overhead is the sum of the overhead generated by checkpointing during the fault-free execution phase and the rework time during the recovery phase.

As shorter is the checkpoint interval smaller is the recovery overhead. On the other hand the application will be interrupted more frequently for checkpointing. This will increase the overhead of checkpointing. The

balance between these two overheads is known as the optimum checkpoint interval.

As the emergent paradigm becomes more and more obvious the following questions also becomes more evident:

— What fault tolerance paradigm offers the smaller overhead to protect a parallel application running on a given parallel computer (considering application memory footprint, communication pattern, and the parallel computer resources)?

— What is the frequency in which uncoordinated checkpoints should be taken?

— How long will be the execution of a parallel application protected by uncoordinated checkpointing?

These statements above can be summarised in one sentence: how to face the emergent paradigm?

1.1 Objectives

The first objective of this work it to answer the three questions presented above. The proposal is to define suitable models that can be used with each paradigm: the coordinated and the uncoordinated. These models should provide an estimation of the application wall time clock running under each fault tolerance paradigm, as well a method to calculate the checkpointing interval.

Considering a parallel applications running on a given parallel computer, those models help users to define which fault tolerant paradigm should be used. A simple comparison between the estimated application execution time when it is being protected by uncoordinated and coordinated checkpointing allows knowing what paradigm introduces less overhead for a specific application running on a given parallel computer.

Moreover, as a second objective, this work intends to provide a methodology to define the variable values used by the models. This permits the use of the proposed models with any fault tolerance architecture.

1.2 Motivations

Computer failures are no longer a rare event but a common problem (22). Currently, there are well-established techniques that provide fault tolerance for parallel applications. To write applications with native support for faults seems to be a good option. However, this approach requires the rewriting of legacy applications. Another solution is to provide fault tolerance at the communication library level and on the parallel environment.

The combination of a resilient parallel environment and a fault recovery technique had been useful in MPI implementations like MPICH (14, 17) and Open MPI (42, 48). However, it is necessary to know how to configure the fault tolerance architecture parameters to make conscientiously use of the parallel machine resources.

Well-defined studies regarding the configuration of fault tolerance architectures are limited to the coordinated paradigm. Moreover, many of these studies are completely theoretical, thus they are far from being the ultimate solution to parallel application users. Another issue which make it difficult to use current models is the definition of the values used by the current checkpoint interval models.

The main motivation of this work is to provide at the same time the knowledge necessary to face the emergent fault tolerance paradigm and make it suitable to be used by parallel applications users.

1.3 Organisation of this Thesis

This thesis contains five chapters. In the Chapter 2 the current paradigm is discussed in details besides the most used fault tolerance protocols in the message-passing systems. Moreover, the technological evolution of state-saving techniques is also presented.

In addition, a view of how parallel computer architectures can be adapted to allows a conscientiously use of machines resources aiming the application fault tolerance. Along this section the reader will found the state of the art in fault tolerance for message-passing applications.

The Chapter 3 presents the contribution of this work. The contribution is a proposal to face the emergent paradigm of fault tolerance in message-passing parallel applications. First of all, a discussion of why current models are not suitable for the emergent paradigm is presented. This chapter starts with the development of the models, and goes with the definition of the variable values of used by models. Finally, the practical uses of the developed models are presented.

In the Chapter 4 the experimental evaluation is presented. The proposed models are compared with models currently available. The experiments focus is to try to minimise the overhead introduced by the fault tolerant tasks for a known fault distribution. The comparison is made focusing on the accuracy of the estimated application execution time and on the precision of the calculated checkpointing interval.

The Chapter 5 states the thesis conclusions and presents the possible future work.

Life has meaning only if one barters it day by day
for something other than itself.
— **Antoine de Saint-Exupéry**

2

Paradigms in Rollback-Recovery Fault Tolerance

This chapter describes the fault tolerance protocols that can be used to protect parallel applications. The description will be made from the overhead perspective. Moreover, the tools that provide support for these protocols, such as checkpoint libraries, will also be described. After that, the boundaries of the current paradigm in rollback-recovery fault tolerance for parallel applications will be stated. Finally, a view of the emergent paradigm will be presented.

2.1 Rollback-Recovery Fault Tolerance Protocols

There are several fault tolerance protocols that can be used with parallel applications. The operation mode of these protocols have been vastly described in literature (34, 50, 62). In this thesis the most used rollback-recovery checkpointing protocols used to provide fault tolerance for message-passing parallel applications would be taken in consideration.

In the following sub-sections the fault tolerance protocols will be analysed from the overhead perspective.

2.1.1 Coordinated Checkpointing

To coordinate checkpoint means that all processes in the parallel application should be checkpointed simultaneously in order to create a global snapshot of the parallel applications (34, 56).

Figure 2.1 is provided to depict the coordinated checkpointing behaviour. In this picture, the grey dashed-line represents the checkpoint coordination limited on both sides by events A and B and E and F. In the time between these events, no communication other the specific coordination communication is allowed. This assures that the global applications snapshot is consistent.

In case of fault, all processes of the parallel application should be rolled back to the last checkpoint. Processes should perform another coordination to ensure that all processes have been recovered before restart execution. In figure 2.1, the numbers on the processes timeline represent different computation instances. As depicted in the figure, all processes lost the computation made between the event B and the event C, in which the fault F crashed the process P_n.

Notice that this kind of protocol treats the parallel application as a set of serial application. This occurs because the protocol synchronises the events of checkpointing and recovering. Thus, all processes stop and continue their execution quasi-simultaneously. However, there are some

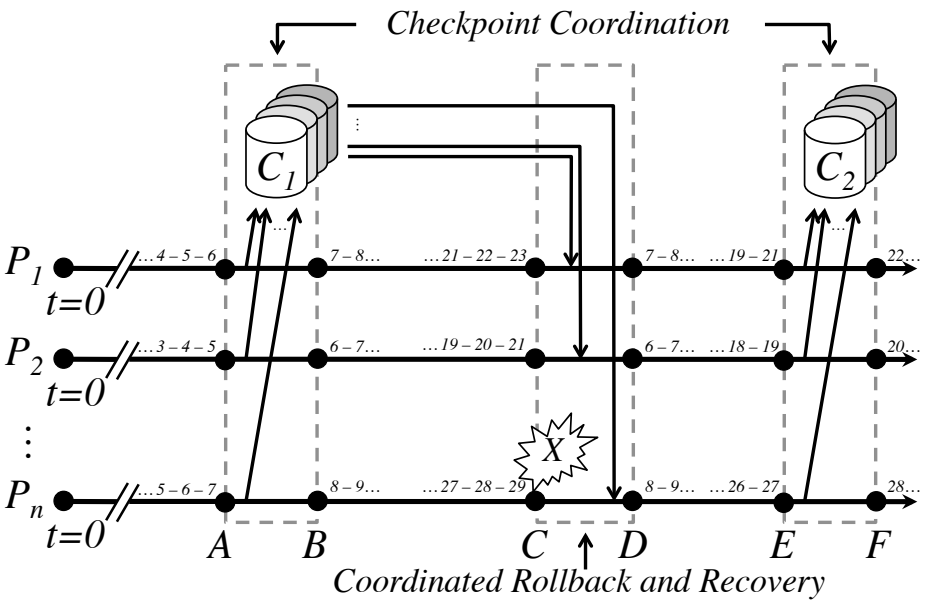


Figure 2.1: Parallel application running with a rollback-recovery fault tolerance assisted by coordinated checkpoints. Work done after the checkpoint and before the fault is lost. Communications are not depicted.

inconveniences on this synchronised behaviour of the coordinated checkpoint. The main inconveniences are:

— If a process requires less time to be checkpointed it should wait the conclusion of the other processes' checkpoint before continues it execution;

— All processes of the parallel application tries to save their state at the same time and concurrently. This means that the storage devices should deal with the load generated by all process at the same time;

— The work done by all processes between the last checkpoint and the fail is lost independent of the number of processes involved in the fail.

Despite of the inconveniences, coordinated checkpoints have been used on many systems. This is the current paradigm in the use of fault tolerance for parallel applications. The main reason of this is because it is quite simple to model its behaviour. Moreover, as an application protected by coordinated checkpointing presents the same behaviour than serial applications, models designed to be used with the second can also be used with the first.

2.1.2 Uncoordinated Checkpointing

As the name suggests, the uncoordinated checkpoint does not require any coordination between processes to perform checkpoints. Checkpoints are taken independently for each process. Figure 2.2 depicts this behaviour.

In figure 2.2 the events A (A_1, A_2, and A_3) are not correlated as well as other events depicted. There is no global snapshot. 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. Other processes continue their execution. However, this behaviour brings pros and cons.

The cons refer to the possibility of creating orphan message on other processes after the recovery of the failed process. An orphan message is created when one process rolls back to a moment previous than the sending or receiving of a message but its counterpart does not roll back with it. This is the reason of storing more than one checkpoint version for each process.

To avoid the domino effect processes with orphan messages should also rollback. Rolling back to previous checkpoint versions may generate a domino effect on the entire parallel application.

To avoid these undesirable situations many authors have proposed methodologies to determine a global consistent state (24, 51, 52). A global consistent state is a set of checkpoints in which neither orphans

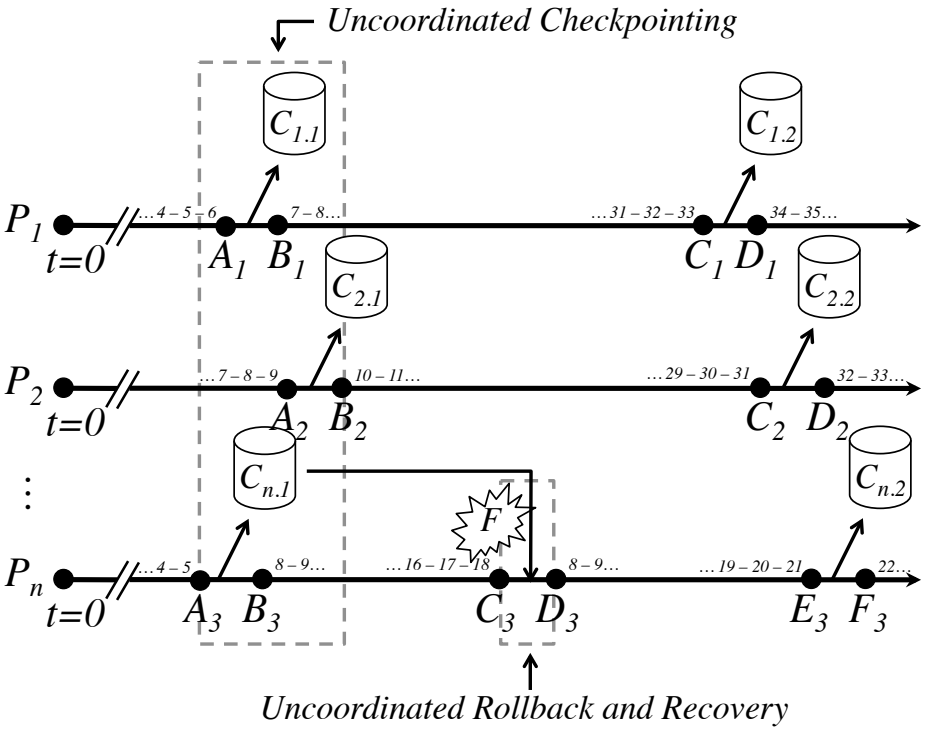


Figure 2.2: Parallel application running with a rollback-recovery fault tolerance assisted by uncoordinated checkpoints. Work done after the checkpoint and before the fault is lost. Communications are not depicted.

messages nor a domino effect exists. There are some models created to define global state that avoids a domino effect (10, 11, 81).

The pros of using uncoordinated checkpoints are related to the fault tolerance performance. As each process does not need to coordinate itself with other, the time need to perform a checkpoint tends to be smaller. Moreover, there is no collective operation involving checkpointing. As the checkpoints do not require to be stored simultaneously, the load in the storage systems tends to be smaller and peaks and bottlenecks could be avoided.

Besides their intrinsic behaviour, another big difference between the uncoordinated checkpoint in its coordinated counterpart concerns the re-

covery phase. If no other process requires to rollback, only the work done by the failed process is lost. This scenario is depicted in figure 2.2. While process P_n restart from the last checkpoint and starts recovering, other processes continue their compute. Moreover, as there is no coordination, processes can take their checkpoints in different moments, according to their requirements.

2.1.3 Event Logging

Event logging fault tolerance protocols are based on the assumption that the state of a process can be reconstructed replaying all process events (15) in the correct order. Even the indeterministic events should be replayed in the same way it has been previously occurred.

The assumption above is founded on the deterministic of the applications (53). The indeterministic events are related to the communication between processes. Thus, if the event logging protocol assures the events replaying in the same order they have originally occurred, it is possible to reconstruct the process state after a fault.

The analysis of the event log overhead depends on the logging protocol and its implementation (4, 18). The event logging can increases the message delivery latency that could generate overhead on the application processes. Moreover, during the recovery phase there are other events associated with the events replaying (72), such as message replay requests.

Figures below depict the operation of the event logging protocol when the log is performed on the sender and on the receiver both for pessimistic and optimistic protocols.

The main difference between the pessimistic and optimistic message logging concerns the moment in which the log storing is performed. Pessimistic logging protocols assure that log data have already been saved

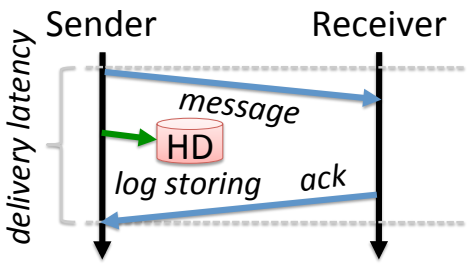


Figure 2.3: Diagram of a pessimistic sender-based message logging protocol.

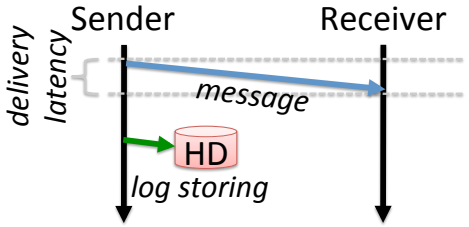


Figure 2.4: Diagram of a optimistic sender-based message logging protocol.

before the delivered message changes other processes state. This behaviour is depicted by the acknowledgement messages in figures 2.3 and 2.5.

On the other hand, optimistic logging protocols do not guarantee that log data have been stored before events change other processes state, an shown in figure 2.4 and 2.6.

From the overhead perspective, optimistic logging protocols introduces less overhead than their pessimistic counterpart. However, in case of fault optimistic logging protocols may require the rollback of non-failed processes in order to replay messages that have not been stored yet. This procedure is necessary to avoid the creation of orphan messages. Additionally, when the logging operation is performed on the sender side the log storing can be overlapped with the message delivery. This overlapping reduces the delivery latency in comparison with the receiver-based protocols.

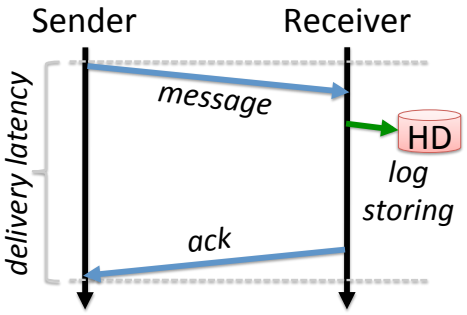


Figure 2.5: Diagram of a pessimistic receiver-based message logging protocol.

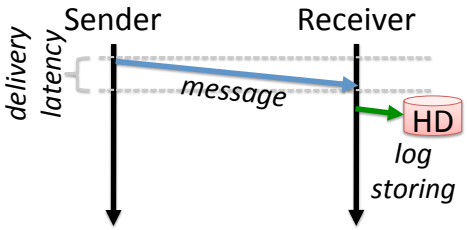


Figure 2.6: Diagram of a optimistic receiver-based message logging protocol.

In which concerns the differences between performing the log on the sender or receiver side the general idea is that the receiver-based event logging doubles the message delivery latency. However, there is another difference concerning the recovery phase. Sender-based event logging protocols require that the sender process replays all messages exchanged with the in-recovery process. On the other hand, for a receiver-based event logging protocol all data necessary to rebuild the state of a failed process is already available on its log repository.

The delivery behaviour of messages in MPI depends on the primitive used for communication. For that reason it may be challenging to determine the overhead introduced by the event logging. It is possible to measure the overhead for specific applications running on certain parallel computer. However, it is not possible to model this behaviour independently of the pair application/machine.

2.1.4 Comparing the Fault Tolerance Protocols

According to the protocol operation mode, different characteristics could be analysed. Some of these characteristics cannot be directly compared. However, the following table depicts some important characteristics of checkpoint-based protocols:

Table 2.1: Summary of some important characteristics of the checkpoint-based fault tolerance protocols for message-passing parallel applications.

Characteristics	Uncoordinated	Coordinated	Communication Induced
Domino effect	Possible	No	No
Orphan processes	Possible	No	Possible
Ages to rollback	Undefined	All processes	Undefined
Recovery protocol	Distributed	Centralised	Distributed
Checkpoints ages	Several	1	Several

As presented in table 2.1 the coordinated checkpointing protocol avoids the domino effect and orphan processes at the same time it need to storage the last checkpoint only. However, this protocol requires the rollback of all parallel application processes.

Other protocols have an undefined behaviour in case of fault. This occurs because communication made with other processes may require to rollback to previous checkpoint other than the last. For this reason these protocols requires the storage of several checkpoints taken at different moments. A key difference between these protocols and the coordinated one concerns the recovery phase. In the uncoordinated and communication induced checkpointing protocols the recovery decision is taken in a distributed manner. This characteristic avoids collective operations to take the decision of what processes should be rolled back.

Besides the checkpointing protocols there are the event logging protocols. Event logging protocols are classified according to the moment

in which the data is stored. The event log data can be stored in three different ways, as shown in table 2.2.

Table 2.2: Summary of some important characteristics of the log-based fault tolerance protocols for message-passing parallel applications.

Characteristics	Pessimistic	Optimistic	Causal
Domino effect	No	No	No
Orphan messages	No	Possible	Possible
Recovery data	Local or Distributed	Local or Distributed	Distributed
Recovery protocol	Local	Distributed	Distributed
Overlap log with message delivery or computation	Possible	Possible	Possible

As shown in table 2.2, recovery data can be stored locally or in more than one process. This depends on the "side" used to perform the event logging. If the event sender is responsible for performing the log, data is stored distributed and the protocol is operating in "sender-base mode". In case of fault, the failed process should request to the event senders the replay of the events. On the other hand, if the event receiver is responsible for performing the log, data is always stored locally. This operation mode is named "receiver-based". If the protocol is operating in sender-based mode, all data necessary to perform the recovery of a failed process is already available locally. In this operation mode other processes do not have to be contacted to replay messages.

All event logging protocols avoid the domino effect. However, the optimistic and casual protocols allow the occurrence of orphan processes because they do not assure that data is already saved before communication events changes processes states. The key characteristic of these protocols is its distributed behaviour. It means that during the recovery phase the failed process do not require any collective operation to perform the recovery. Decision are taken locally or in a distributed manner.

Other characteristics and differences between these logging protocols are vastly available in literature (5).

Currently, there are many MPI implementation that provide fault tolerance for parallel applications, such as the MPICH-V project (14, 17, 19), Open MPI (48), Egida (71), MPI-FT (59), FT-MPI (37, 38), MPI/FT (12, 13), Starfish (1), LA-MPI (9) and LAM/MPI (75).

2.1.5 The Checkpoint Interval

The checkpoint interval is the time between two checkpoint instances. If the time needed to perform a checkpoint is negligible the best strategy is to perform a checkpoint after every instruction. However, the time needed to perform a checkpoint depends on the application memory footprint and the throughput of the storage devices, besides other operations intrinsic to the checkpointing protocol.

The optimum checkpoint interval defines the frequency in which checkpoint should be taken to introduce the minimum overhead on the application. This overhead is the sum of the overhead generated by checkpointing during the fault-free execution phase and the rework time (the amount of work completed after a checkpoint and prior to a failure) during the recovery phase.

As shorter is the checkpoint interval smaller is the recovery overhead. On the other hand the application will be interrupted more frequently for checkpointing. This will increase the overhead of checkpointing. The balance between these two overheads is known as the optimum checkpoint interval.

The use of analytical models to define the checkpoint interval for serial applications has been studied from the 70's until today. In 1974, Young (86) introduced the first order approximation, an analytical model to determine the checkpoint interval.

Using Young's model it is possible to calculate the checkpoint interval once the user knows the time needed to perform some fault tolerance tasks and the system fault probability. More recently, Gropp (44) presented a simpler model achieving a similar result for the checkpoint interval. Nevertheless, Gropp uses a different approach to deduce his model.

Many years after Young, Daly presented a very deeply analytical study (29) to determine the higher order estimation of the checkpoint interval. Daly analyses different scenarios such as multiple failures between checkpoints, fractional rework, failures during restarts *et cetera*. The model achieved by Daly presents the same structure of Young's model but is more precise.

Other strategies to calculate the checkpoint interval have been proposed. Some of these proposals concerns the definition of maximum and minimum values for the checkpoint interval (68), the use of variational calculus (57), or strategies based on the application algorithm (27, 84).

Despite the importance of the uncoordinated checkpoint protocols, as far as we know, there is no model to calculate the checkpoint interval that minimises the fault tolerance overhead for these protocols. Furthermore, models designed to be used with serial applications and also used with coordinated checkpoints may be not appropriate for uncoordinated checkpointing protocols (41).

Also, it is important to notice that values different than the optimum checkpoint interval increases the fault tolerance overhead (49).

2.2 Evolution of the Technology

As parallel computers start to grow in number of compute nodes and in total memory size users notice that the fault tolerance overhead grows as well. Thus, there are many proposals to reduce the overhead introduced by the fault tolerance tasks. These proposals concern new technologies

for checkpointing, better logging strategies, and small modifications on the parallel computer architecture.

2.2.1 Parallel Computer Performance and Resources

The throughput of the storage device plays a major role in the time needed to perform a checkpoint. It is notorious that the total memory size of a parallel machine grows faster than its storage throughput.

Some authors believe that future generations parallel computers should provide storage support exclusively to fault tolerance, such as solid state drive devices (22). Besides the support of additional storage devices, lightweight file systems have been developed specifically for checkpointing purposes (63).

Despite those architectural modifications on the parallel computer, saving the whole parallel application state into a storage system represents a huge load agains these devices. To solve this peak problem parallel computers designers may over-dimension the storage system. However, when the application is not being checkpointed the storage system is practically idle.

2.2.2 Checkpointing Techniques

There are many checkpoint libraries and tools that can be used to support the checkpointing of parallel applications, such as BLCR (47), DMTCP (7), CLIP (26), and CRAK (88). These are kernel level or user space level libraries that save the program memory state in a checkpoint file. The file generated by these kind of libraries are equivalent to the process memory footprint.

In order to reduce the time needed to perform a checkpoint many strategies have been developed (74). These strategies concerns the creation of in memory checkpoint (87), incremental checkpointing (43, 61),

non-blocking checkpointing (28), and compiler-assisted checkpointing (55, 58).

Other optimisations on the checkpointing performance concerns the moment in which the checkpoint is taken. Some strategies try to perform the checkpointing opportunistically (8), reducing the storage contention. Other strategies uses information from the application programmer, the compiler, and the run-time system to decide when a checkpoint should be taken (65, 66).

2.2.3 Logging Techniques

To reduce the event logging overhead many work have been done. All these researches aim the reduction of the event logging latency. However, it is a common sense that the event logging is the major root cause of the fault tolerance overhead. There are work that modifies the event logging model to increase the overlap of the logging with the message delivery (16). Other authors have been working on re-designing the storage procedure of the pessimistic logging (76).

It is possible to manage the optimism degree in order to minimise the latency introduced by the event logging (73). Non-blocking and orphan free event logging is also possible (3). However, there are different reasons to choose one logging protocol instead of other (21). The decision of what event logging protocol should be used depends on the performance and the desired reliability.

2.3 Boundaries of the Current Paradigm

Despite of the achievements on the reduction of the state-saving techniques, the coordinated checkpointing protocol still has many scalability issues. These issues increases according to the size of the parallel computer (45, 67).

It is possible to know the overhead of checkpointing a parallel application on large scale machines (64, 82), and it is notorious that to coordinate and to store the state of huge parallel applications represent a high overhead on the execution of these applications.

In many situations the use of event logging could be better than coordinated checkpointing (20, 54), *e.g.* huge parallel applications with a small number of messages exchanged between processes. The main root cause of this resides on the poor scalability of the coordinated checkpointing (36).

2.4 Beyond the Current Paradigm

If the current paradigm in fault tolerance for parallel applications does not introduces a small overhead, how to efficiently protect these application running on the next-generations machines? Many authors have pointed a new approach (22, 23, 30, 35).

This new approach tries to combine the use of event logging with uncoordinated checkpointing (31, 32, 85). Indeed, there are MPI libraries that support many fault tolerance protocols (19).

The idea behind these approaches consists of combine the benefits of checkpoint and logging protocols. Bearing scalability, the uncoordinated checkpointing protocol provide better results (2, 69). To avoid the undesirable situations that this checkpointing protocols brings, it is necessary to combine the uncoordinated checkpointing with an event logging protocol.

The combination of uncoordinated checkpoint and an efficient message logging reduces the amount of memory required for event logging (6). This is the emergent fault tolerance paradigm that can provide scalable fault tolerance for parallel application on the future parallel computers.

2. PARADIGMS IN ROLLBACK-RECOVERY FAULT TOLERANCE

The machine does not isolate man
from the great problems of nature
but plunges him more deeply into them.
— **Antoine de Saint-Exupéry**

3 Propose to Face the Emergent Paradigm

In this chapter the proposal to face the emergent paradigm in fault tolerance will be presented. To use uncoordinated checkpoint combined with event logging users should be capable to define the checkpoint interval for this fault tolerance protocol. Moreover, it is good to know the overhead that this protocol introduces on the parallel application execution time.

The proposal concerns mathematical models to define the checkpoint interval and the overhead introduced by fault tolerance tasks. As explained before, the use of current models, which have been designed to be used with coordinated checkpoint, may not be adequate for uncoordinated checkpointing.

The modifications necessary to use these models with uncoordinated checkpointing will be explained. These modifications include the definition of a factor that represents the inter-dependence existent between different processes in parallel applications.

3.1 What is Missing in Current Fault Tolerance Models?

Current checkpoint interval models have been developed to define the checkpoint interval that minimises the overhead introduces by fault tolerance tasks. These tasks can be divided into protection and recovery tasks. The first represents procedures performed to save the application state while the second represents procedures performed after a fault.

While using coordinated checkpoint all processes are rolled back and start recovery in case of a fault. This coordinated behaviour makes parallel applications perform similar to serial applications.

For the sake of easiness, the deduction of the checkpoint interval model for parallel applications protected by coordinated checkpointing will also be explained using a serial application but considering the event logging and the relationship between processes.

Analysing current checkpoint interval models, which have been designed for serial applications and considering fault tolerance tasks performed between two faults the overhead introduced by these tasks can be defined as:

$$Overhead_{serial} = T_c + T_r \tag{3.1}$$

where T_c is the time needed to perform all protection procedures and T_r is the total time needed to perform all recovery procedures.

In order to better understand the deduction of the serial application fault tolerance model let us consider figure 3.1 and the following naming system:

t_c as the time spent on a checkpoint operation including the storage time. In other words, it is the application interruption time necessary to take a checkpoint.

t_d as the time needed to detect a fault, also known as fault detection latency.

t_l as the time needed to load a checkpoint from storage.

t_r as the amount of time needed to recover a failed process and achieve the computation point just before fault. It is the reworking of the previous lost computation, also known as fractional rework.

Q as the quantity of checkpoints that should be performed between two faults.

T_c as the total protection time represented by the sum of all t_c between two faults. This value can be obtained multiplying t_c by Q.

T_r as the total recovery time per fault represented by the sum of t_d, t_l, and t_r.

T_s as the total time to run the application without faults and any sort of fault tolerance.

α as the mean time to interrupt (MTTI) for a given system, which is the inverse of the fault probability.

σ as the checkpoint interval used to run the application. It can also be considered as the useful time for the application to compute.

As shown in figure 3.1, the recovery task occurs at the beginning of the period between faults F_x and F_y. Recovery takes T_r time (segment $\overline{BE}$) to conclude and after this comes one or more computational segments followed by checkpoints. Each checkpoint requires t_c time (segments $\overline{EF}$, $\overline{GH}$, and $\overline{IJ}$) to be taken.

Checkpoints are separated by an application computational period represented by σ (segments $\overline{FG}$ and $\overline{HI}$). Ergo, σ is the interval between checkpoints, *ipso facto* the checkpoint interval. Segment $\overline{JK}$ will be lost

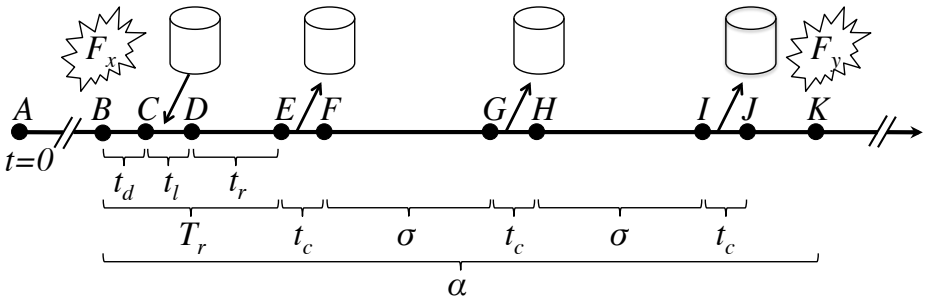


Figure 3.1: Between faults F_x and F_y there is a recovery time (T_r) and three checkpoints (t_c) among computational periods (σ).

due to fault F_y. This work will be redone after the next recovery phase (not depicted in figure 3.1).

As aforementioned, T_c is the sum of all t_c, it represents the sum of segments $\overline{EF}$, $\overline{GH}$, and $\overline{IJ}$ in figure 3.1. Time spent on protection and recovery tasks is not useful application time, thus these tasks are considered to be overhead. This assumption leads us to equation 3.1

3.1.1 Developing the Model

As mentioned previously, T_r is the sum of the fault detection latency (t_d), checkpoint loading from storage (t_l) and the fractional rework (t_r). It can also be seen in figure 3.1 and leads us to the following equation:

$$T_r = t_d + t_l + t_r \tag{3.2}$$

As proved by Daly (29), it is accepted to assume that interrupts occur halfway through the checkpoint interval. Thus, it is valid to consider the fractional rework (t_r) as half of the checkpoint interval (σ). Rewriting equation 3.2 in the terms above it gives us the following:

$$T_r = t_d + t_l + \frac{\sigma}{2} \tag{3.3}$$

The same demonstration can be used in relation to the fault detection latency. However, there are many fault detection mechanisms that can be used. Supposing that a system uses a heartbeat/watchdog mechanism, the value of t_d is half of the frequency used to configure such a mechanism. Because it is a user-defined value let consider it as a system variable. Thus, the model maintains its original aspect presented in equation 3.3.

The total time spent on process protection (T_c) depends on the checkpoint frequency. In order to calculate T_c, which is the sum of all t_c, the number of checkpoints (Q) should be defined.

The number of checkpoints (Q) can be defined as the number of segments composed of checkpoint (t_c) and useful computing time (σ) that fits the period between faults (α), excluding the recovery time. This assumption is reflected in the equation below:

$$Q = \frac{\alpha - T_r}{\sigma + t_c} \tag{3.4}$$

As shown in figure 3.1, the work done after the last checkpoint and just before the fault F_y is lost. This lost work, represented in figure 3.1 by the segment $\overline{JK}$, will be redone during next recovery phase; in addition t_r represents the recuperation of the work lost before fault F_x. Thus, segment $\overline{JK}$ has already been included in the model.

With the number of checkpoints already defined, it is easy to calculate the total protection time as shown below:

$$T_c = Q * t_c \tag{3.5}$$

Using equations 3.3, 3.4 and 3.5 and applying some algebraic operations on the preliminary overhead model presented in equation 3.1 the following overhead equation is obtained:

$$Overhead_{serial} = \frac{\sigma^2 + 2(\sigma t_d + \sigma t_l + \alpha t_c)}{2(\sigma + t_c)} \tag{3.6}$$

It is possible to find the value of σ_{opt} that minimises the fault tolerance overhead by deriving the overhead equation 3.6 with respect to σ and setting the result to zero. Considering the positive solution, this operation brings us to the optimum checkpoint interval hereunder:

$$\sigma_{opt} = \sqrt{t_c^2 - 2t_c t_d - 2t_c t_l + 2\alpha t_c} - t_c \tag{3.7}$$

The checkpoint interval model that minimises the fault tolerance overhead presented in equation 3.7 includes the failure detection latency. This additional parameter distinguishes this model from those of Young (86), Gropp (44), and Daly (29). Nevertheless, supposing a scenario in which the fault detection latency tends to zero and the time needed to perform a checkpoint is equal to the time needed to load a checkpoint from storage, the models are very close, as shown in the following equation:

$$\sigma_{opt} = \sqrt{2\alpha t_c - t_c^2} - t_c \tag{3.8}$$

A comparison between this model and those of Young (86), Gropp (44), and Daly (29) is presented in table 3.1. This table shows a summary of the models under the nomenclature used in this paper. For comparison purposes the fault detection latency has been defined to zero and the time needed to perform a checkpoint have been considered equal to the time needed to load it from storage.

Table 3.1: Checkpoint interval models under the same nomenclature.

Author	Optimal Checkpoint Interval
Young	$\sqrt{2\alpha t_c}$
Gropp	$\sqrt{2\alpha t_c}$
Daly	$\sqrt{2\alpha t_c} - t_c$
Fialho	$\sqrt{2\alpha t_c - t_c^2} - t_c$

3.1.2 The Cost Function

The cost function allows the estimation of the application run time running with fault tolerance and a given fault distribution.

Considering T_s as the total time to run a given application without faults and any sort of fault tolerance and T_{est} as the estimated time to run the application with fault tolerance, the cost function can be represented by the following equation:

$$T_{est} = T_s \left(1 + \frac{Overhead}{\alpha}\right) \tag{3.9}$$

Where the cost is defined by the original application run time plus the overhead defined in equation 3.6. Replacing equation 3.6 in 3.9, the further cost function is achieved:

$$T_{est} = T_s \left(1 + \frac{\sigma^2 + 2(\sigma t_d + \sigma t_l + \alpha t_c)}{2\alpha(\sigma + t_c)}\right) \tag{3.10}$$

Different than the checkpoint interval models, the cost functions presented by those author are not similar, as shown in table 3.2.

3.2 The Inter-Process Dependency Factor

The main problem of using the checkpoint interval model explained above with uncoordinated checkpointing protocols refers to the fact that faulty

Table 3.2: Cost functions of checkpoint interval model according to each author.

Author	Cost Function
Young[1]	$\alpha - \frac{\sigma}{1-e^{(\sigma+t_c)/\alpha}}$
Gropp[2]	$\frac{T_s}{\sigma}\left(t_c + \sigma + \frac{\sigma t_l + \frac{\sigma^2}{2}}{\alpha}\right)$
Daly	$\alpha e^{t_l/\alpha}(e^{(\sigma+t_c)/\alpha} - 1)\left(\frac{T_s}{\sigma} - \frac{t_c}{\sigma+t_c}\right)$
Fialho	$T_s\left(1 + \frac{\sigma^2+2(\sigma t_l+\alpha t_c)}{2\alpha(\sigma+t_c)}\right)$

process should be rolled back while other processes continue their execution as shown in figure 3.2.

All processes in the parallel application can fail. However, each process failure may impact other processes in a distinct way. This means that if process n fails, one or more processes can hang waiting for the recovery of n to be completed, *e.g.* considering a master/worker application, if the master process fails, all workers may wait for the recovery of the master.

In figure 3.2 process P_1 had to wait the recovery of process P_n before sending it the message marked with the continuous red line. On the other hand, process P_2 did not had to wait, because this process does not communicate with process P_n.

This means that processes in a parallel application have an intrinsic inter-dependent relationship. This relationship is defined by the messages exchanged between processes. In this thesis the relationship existent between processes will be named *inter-processes dependency factor*.

[1]This function calculates the overhead *per* fault.

[2]The original text refers to "k_1 = cost to read and restore checkpoint" (44). It is not clear if this variable considers the fractional rework time or not. Assuming that the authors do not consider it, table shows a re-written version of the model to a common nomenclature and replacing the original term k_1 by t_l.

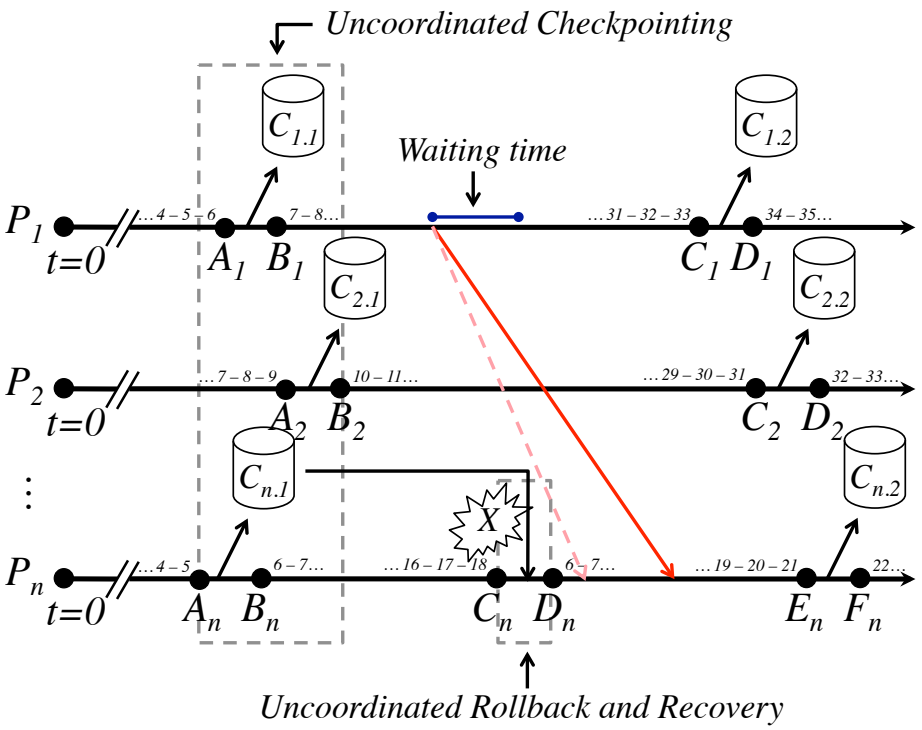


Figure 3.2: Rollback and recovery behaviour of the uncoordinated checkpointing protocol.

3.3 A Model for the Emergent Fault Tolerance Paradigm

To create a fault tolerance model for parallel applications, first of all, we should introduce some statements of parallel application behaviour.

The first statement concerns the disturbances added by event logging operations and how failures impact on different application processes. These disturbances lead to changes in our previous model.

The other statement refers to the rollback of only one of all the processes that compose parallel applications. This behaviour is defined by the inter-process dependency factor.

Another important issue refers to the frequency in which processes

should be checkpointed. The optimum checkpoint interval for each process depends on characteristics of the process itself. This means that different processes in the parallel application may have different checkpoint intervals.

Moreover, the checkpoint interval for a specific process may change during the application execution if changes occur in the application characteristics, such as memory footprint, or in the parallel machine status, such as the network latency. The use of different checkpoint intervals for each application process is only possible in uncoordinated checkpoint protocols.

The event logging disturbance depends on the logging protocol used (34). In this thesis two of these protocols will be analysed: pessimistic sender-based and pessimistic receiver-based. The reason of choosing pessimistic protocols is because these protocols avoid the undesirable domino effect and the creation of orphan processes.

Event logging protocols can introduce overhead on the parallel application execution in two different situations: one refers to the fault free operation and the other to the recovery phase.

Independently of the protocol used, event logging leads to changes in the time needed to recover a failed process (T_r) and the time needed to protect a process (T_c).

To better understand the the event logging overhead let us consider figures 3.3 and 3.4 and the addition of two new terms to the naming system presented above:

Δ_{lp} as the sum of all time added to message delivery due to the logging procedure.

Δ_{lr} as the sum of all time spent no processing the message log after a fault. The majority of this is the replaying time, if it exists.

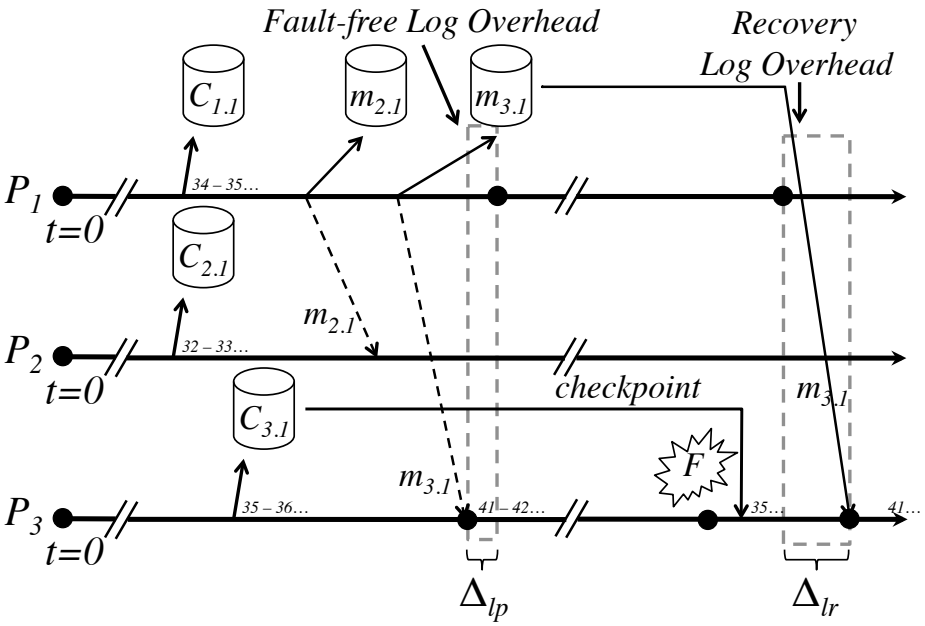


Figure 3.3: Overhead introduced by the sender-based message logging protocol for protection (Δ_{lp}) and for recovery (Δ_{lr}).

When the sender processes perform the event logging operation, they should manage the data storage. This protocol allows message logging to be executed in parallel with the message delivery. Thus the overhead introduced by the logging procedure (Δ_{lp}) tends to be small or even nonexistent, *e.g.* the message $m_{2.1}$ in figure 3.3.

During the recovery phase, sender processes should replay messages sent to the faulty process. Thus, the overhead introduced by this protocol during recovery (Δ_{lr}) represents the time spent on replaying messages.

Figure 3.3 depicts the overhead introduced by this protocol during a fault-free execution and during the recovery phase. As shown, the recovery of process P_3 depends on other processes.

When the message logging operation is performed on the receiver process, as shown in figure 3.4, the data storage cannot be overlapped with message delivery. In general, receiver based logging doubles the time

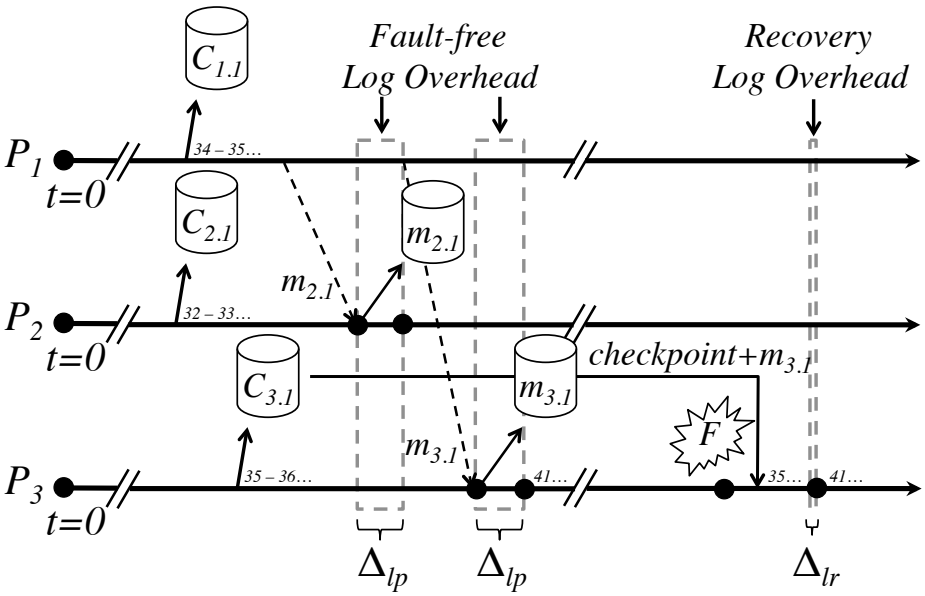


Figure 3.4: Overhead introduced by the receiver-based message logging protocol for protection (Δ_{lp}) and for recovery (Δ_{lr}).

needed for message delivery. This occurs because data should be saved in storage after message delivery.

However, in case of faults, only the faulty process is involved in recovery. Since messages do not need to be replayed, the time needed to process the message log (Δ_{lr}) tends to be unappreciable and there is no message replaying time.

3.3.1 Developing a New Model

Modifying the aforementioned equations 3.3 and 3.5 in order to reflect the message logging overhead, leads us to the following equations:

$$T_r = t_d + t_l + \frac{\sigma}{2} + \Delta_{lr} \tag{3.11}$$

$$T_c = (Q * t_c) + \Delta_{lp} \tag{3.12}$$

With regard to the disturbance introduced by faults; this depends completely on application characteristics like programming paradigm, communication pattern, data distribution, *et cetera*. However, the inter-process dependency factor affects the aforementioned T_r lowering its weight on the overhead equation.

Rewriting the overhead equation 3.1 in order to consider this assertion, the following equation is produced:

$$Overhead_{parallel} = \phi(T_r) + T_c \tag{3.13}$$

where ϕ represents the inter-process dependency factor.

Regarding this factor (ϕ), small values represent less dependency between processes while the higher value means that when one process fails all other processes should wait for the recovery of the failed process.

The upper limit to the inter-process dependency factor is 1. In this case the models for parallel and serial applications become the same. The lower limit depends on the number of processes in the parallel application and the relationship existent between them.

Replacing equations 3.11 and 3.12 in equation 3.13 and after applying some algebraic operations, the following overhead equation is obtained:

$$\begin{aligned} Overhead_{parallel} = & \frac{\phi\sigma^2 + \sigma(2\phi t_d + 2\phi t_l + \phi t_c + 2\phi\Delta_{lr} - t_c + 2\Delta_{lp})}{2\sigma + 2t_c} + \\ & \frac{2t_c(\phi t_d + \phi t_l + \phi\Delta_{lr} + \alpha - t_d - t_l - \Delta_{lr} + \Delta_{lp})}{2\sigma + 2t_c} \end{aligned} \tag{3.14}$$

It is possible to find the value of σ_{opt} that minimises the fault tolerance overhead for parallel applications by deriving equation 3.14 with respect to σ, and setting the result to zero. Considering the positive solution, this operation brings us to the following optimum checkpoint interval:

$$\sigma_{opt} = \frac{\sqrt{\phi t_c(t_c + 2\alpha - 2t_d - 2t_l - 2\Delta_{lr})}}{\phi} - t_c \tag{3.15}$$

Supposing a system whose detection latency tends to zero and the time needed to perform a checkpoint is comparable to the time needed to load a checkpoint from storage, the equation 3.15 can be simplified to:

$$\sigma_{opt} = \frac{\sqrt{\phi t_c(2\alpha - t_c - 2\Delta_{lr})}}{\phi} - t_c \tag{3.16}$$

3.3.2 Defining The Inter-Process Dependency Factor

Considering a parallel application protected by an uncoordinated checkpointing as the study scenario. In case of fault of one process, other processes may be affected if they need to communicate with the faulty process.

Figure 3.5 shows an execution of this application in a fault-free scenario, while figure 3.6 represents the same execution in case of fault. For the sake of easiness the message logging operation has been omitted.

Processes in this example application exchange a message on the computation time 20. In figure 3.5 the communication is performed according to the expected. However, in figure 3.6 because of process P_1 has rolled back process P_2 have to wait for process P_1 achieve the computation time 20 before continue its execution.

As aforementioned, the inter-process dependency factor tries to model the dependency between all processes in the parallel application. This de-

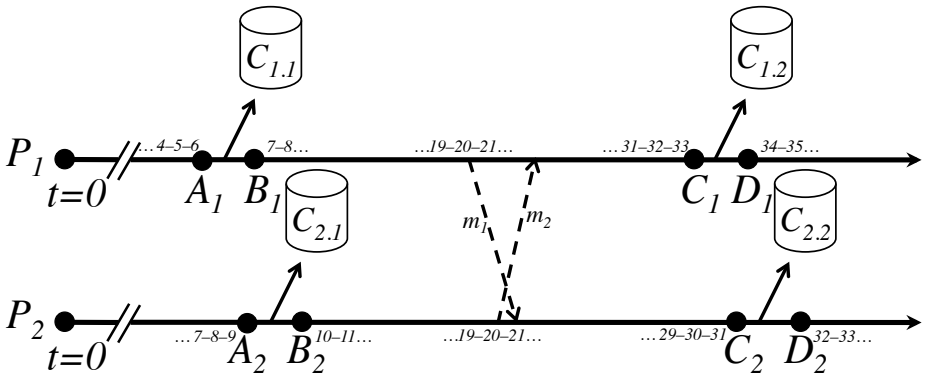


Figure 3.5: Parallel application running with a rollback-recovery fault tolerance architecture assisted by uncoordinated checkpointing in a fault-free scenario. The message logging operation has been omitted.

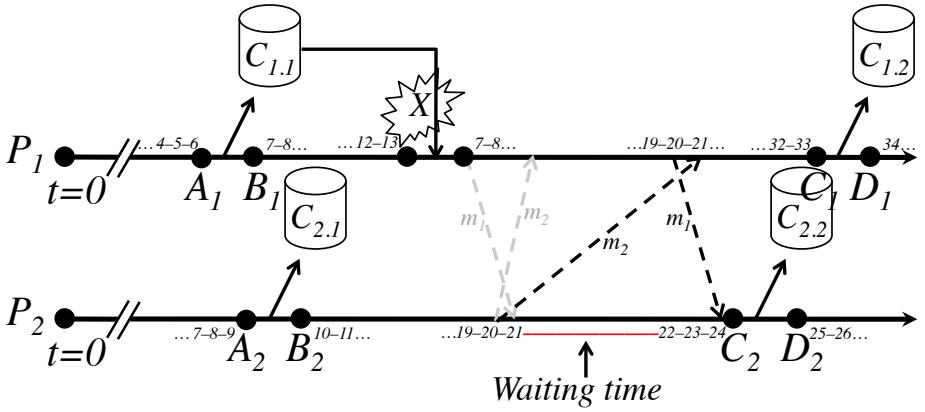


Figure 3.6: Parallel application running with a rollback-recovery fault tolerance architecture assisted by uncoordinated checkpointing in a faulty scenario. The message logging operation has been omitted.

pendency exists because of the communication exchanged between these processes. The function $P(n)$ should consider the number of processes which may be affected by a fault in a given process.

As communication defines the relationship between processes, to define the $P(n)$ function the application's communication characteristics should be considered.

There are three main communication characteristics and one faulty process characteristics that should be considered. Regarding communication characteristics there are: peers (sources and destinations), frequency, and the recursive effect caused by a fault (process B hangs because of process A, thus process C hangs because of process B, ...). Regarding the faulty process characteristic, is the time needed to recover a faulty process.

In this first approximation to the $P(n)$ function only the peers which the process communicates with plus the faulty process will be taken in consideration. The following equation shows the definition of the global inter-process dependency factor using such analysis:

$$\phi_{global} = \frac{\sum_1^N P(n)}{N^2} \tag{3.17}$$

where $P(n)$ is the function which defines the number of processes that communicate with the process n plus itself, and N is the total number of processes in the parallel application.

Based on the master/workers application used as example above, let us assume an application running with 8 processes in which workers do not communicate among themselves. And, supposing a function $P(n)$ which considers the existence of communication as the only dependency between processes. If the master process fails all workers should wait for its recovery, then $P(master)$ is 8.

If any worker fails just the master may wait for it, then $P(worker)$ is 2 for all 7 workers. In compliance with this assumption the dependency factor for this application is 0.34375, as shown in table 3.3. It is a very simple method to define the global inter-process dependency factor.

The equation 3.17 reflects this relationship of all processes in an application. However, applications are composed of different phases (83). This

Table 3.3: Values of the inter-process dependency factor for a Master/Worker application running with 8 processes using the first approximation to define $P(n)$.

Process	**Peers**	P_n	ϕ
Master	8	8	1
Worker_1	2	2	0.25
Worker_2	2	2	0.25
...	...	...	...
Worker_7	2	2	0.25
Global			0.34375

means that during the application execution processes may have different relationships according to the phase it is running.

The best approach is to define the inter-process dependency factor individually for each application process. For that, the following model has been developed:

$$\phi_{individual} = \frac{P(n)}{N} \tag{3.18}$$

Different from the previous model for the ϕ, this new one does not reflect the inter-process dependency of the whole application but the relationship of each process with the others, giving each process an individual value for the inter-process dependency factor. However, the sum of the values of ϕ of all processes divided by the number of processes that compose the parallel application is equal to the value of the global ϕ.

3.3.3 The Cost Function

Considering T_p as the total time to run a given parallel application without faults and any sort of fault tolerance and T_{est} as the estimated time to run the application with fault tolerance, the cost function can be represented by the following equation:

$$T_{est} = T_p \left(1 + \frac{Overhead}{\alpha}\right) \tag{3.19}$$

where the cost is defined by the original application run time plus the overhead defined in equation 3.14. Replacing equation 3.14 in 3.19, the further cost function is achieved:

$$T_{est} = T_p \Bigg[1 + \frac{\phi\sigma^2 + \sigma(2\phi t_d + 2\phi t_l + \phi t_c + 2\phi\Delta_{lr} - t_c + 2\Delta_{lp})}{\alpha(2\sigma + 2t_c)} + \frac{2t_c(\phi t_d + \phi t_l + \phi\Delta_{lr} + \alpha - t_d - t_l - \Delta_{lr} + \Delta_{lp})}{\alpha(2\sigma + 2t_c)}\Bigg] \tag{3.20}$$

3.4 Acquiring Values for the Model's Variables

Current models to calculate de optimum checkpoint interval are far from being the ultimate solution to the checkpoint interval. The major root causes of this reside in the definition of the variables value used by these models besides the use of simplified models. The use of average values as input parameters for models reduces their accuracy.

During the execution, some application characteristics may change over the time as well as the parallel machine status. Thus, models will experience a loss of accuracy because the checkpoint interval does not change to reflect such changes.

Models variables depend on the application characteristics such as the memory footprint and the communication pattern, besides the system load such as the storage and the communication network.

This thesis aims to propose a methodology to define in run-time the checkpoint interval for parallel applications. The dynamic definition relies on the measuring of the time spent on fault tolerance tasks to obtain

values for the checkpoint interval model variables. It turns the checkpoint interval model versatile enough to accommodate changes in the application characteristics throughout its execution.

Our propose to define the checkpoint interval in run-time rests on two foundations: first on a checkpoint interval model and second on the measurement of the time needed to perform fault tolerant tasks. The second provides the values for the variables used by the first.

To define the value of these variables, a monitoring mechanism of the fault tolerant tasks performed during application execution is used. The diagram shown in figure 3.7 depicts such a mechanism.

The time needed to perform a checkpoint operation (t_c) is measured by the timer depicted in events 1 and 2 of the diagram. The inter-process dependency factor is calculated by analysing sources and destinations of messages exchanged with other processes. It is depicted in the diagram by event 4.

The message logging overhead depends on the logging protocol used (34). The time added to message delivery due to the logging procedure (Δ_{lp}) is measured by the timer depicted in events 3 and 4. When the message logging operation is performed on the receiver process, as shown in figure 3.4, in case of faults, only the faulty process is involved in recovery. Since messages do not need to be replayed, the time needed to process the message log (Δ_{lr}) tends to be unappreciable. Thus, the value of the Δ_{lr} variable can be considered as zero (72).

The time needed to load a checkpoint (t_l) cannot be measured using the proposed methodology if no fault occurs. However, as a first approximation it is valid to consider this time equal to the time needed to perform a checkpoint. It does not reduces the checkpoint interval model accuracy because variables related to the recovery phase, with the exception of the rework time, tend to be inappreciable (86).

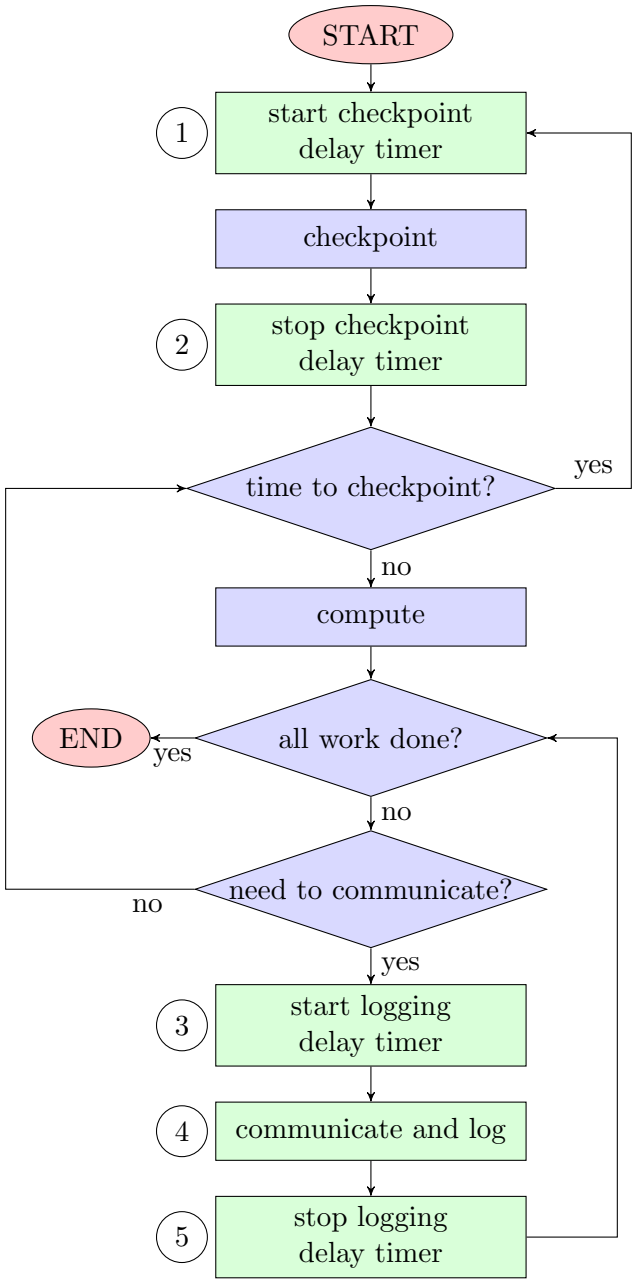


Figure 3.7: Diagram of the methodology used to define model variables values in run-time.

After all variables values needed by the checkpoint interval model had been already defined in run-time it is possible to use the checkpoint interval model defined in section 3.3.1 to calculate the checkpoint interval.

3.5 Exploiting the Solution

With the checkpoint interval model and the methodology to acquire the model's variables this chapter intends to answer the questions raised in the introduction of this thesis:

— What fault tolerance paradigm offers the smaller overhead to protect a parallel application running on a given parallel computer?

— What is the frequency in which uncoordinated checkpoints should be taken?

— How long will be the execution of a parallel application protected by uncoordinated checkpointing?

To know what fault tolerance protocol should be used to protect a parallel application it is necessary to know *a priori* how the fault tolerance tasks perform on the target parallel computer. If the values of the variables used on the checkpoint interval model are already known it is possible to use the cost function of both models to make a simple comparison.

However, if there is no prior knowledge about the performance of the fault tolerance tasks on the target parallel computer, the methodology to found the values of the variables used in models can be useful. Both models are short enough to be included in any MPI library with fault tolerance capabilities. Thus, a multi-protocol MPI library implementation such as MPICH-V (19) can be used to protect the parallel application.

The matter in question about multi-protocol MPI implementations concerns the possibility to dynamically select the fault tolerance protocol. As far as the knowledge of the author, there is no published work about changing in run-time the fault tolerance protocol for parallel applications. Thus, the only way to select the fault tolerance protocol is before application launching.

The second question concerns the moment in which checkpoints should be taken while using an uncoordinated checkpointing protocol. The checkpoint interval model defined in equation 3.15 answers this question. As well, the next question which concerns the overhead introduced by fault tolerance is answered by the cost function defined in equation 3.19.

3.5.1 Recovery Time Constraints

Additionally, the model can be used to guarantee the maximum recovery time per fault. For that a small modification should be made on the proposed model. The time to recovery (T_r) should be replaced by the user defined value for the *maximum time to recovery* – MaxTTR.

To satisfy the user requirements, the checkpoint interval model as well as the cost function should be re-wrote as follows:

$$MaxTTR_{user} = \sigma + t_l + t_d + \Delta_{lr} \tag{3.21}$$

where $MaxTTR_{user}$ represents the *maximum time to recovery* specified by the user. Considering this equation, the calculation of the maximum value that should be used for the checkpoint interval becomes easy:

$$\sigma_{max} = MaxTTR_{user} - t_l - t_d - \Delta_{lr} \tag{3.22}$$

Considering the recovery time constraint the fault tolerance overhead should be:

$$Overhead_{parallel} = \frac{2\phi\sigma^2 + \sigma(2\phi t_d + 2\phi t_l + 2\phi\Delta_{lr} + 2\phi t_c - t_c + 2\Delta_{lp})}{2\sigma + 2t_c} + \frac{2t_c(\phi t_d + \phi t_l + \phi\Delta_{lr} + \alpha - t_d - t_l - \Delta_{lr} + \Delta_{lp})}{2\sigma + 2t_c} \tag{3.23}$$

Then, the cost function for a parallel application protected by uncoordinated checkpointing with recovery time constraint is:

$$T_{est} = T_p\left[1 + \frac{2\phi\sigma^2 + \sigma(2\phi t_d + 2\phi t_l + 2\phi\Delta_{lr} + 2\phi t_c - t_c + 2\Delta_{lp})}{2\sigma + 2t_c} + \frac{2t_c(\phi t_d + \phi t_l + \phi\Delta_{lr} + \alpha - t_d - t_l - \Delta_{lr} + \Delta_{lp})}{2\sigma + 2t_c}\right] \tag{3.24}$$

3.5.2 Heterogeneous Processes on Parallel Applications

Many parallel applications are composed by processes that perform different tasks. This behaviour is common on parallel programming paradigms such as Master/Worker, Workflow, and Map/Reduce.

The processes of applications designed under these paradigms may present different memory footprint and communication pattern. It is obvious that the fault tolerance tasks may perform differently according to each process.

In this proposal the checkpoint interval is defined per process, as well as the methodology used to acquire variable values. This means that during the application execution each process can dynamically define the checkpoint interval.

The dynamic definition of the checkpoint interval permits the adaptation of the checkpoint interval to changes on the application characteristics and on the parallel computer status. Figure 3.8 depicts how the checkpoint interval can be re-defined in run-time.

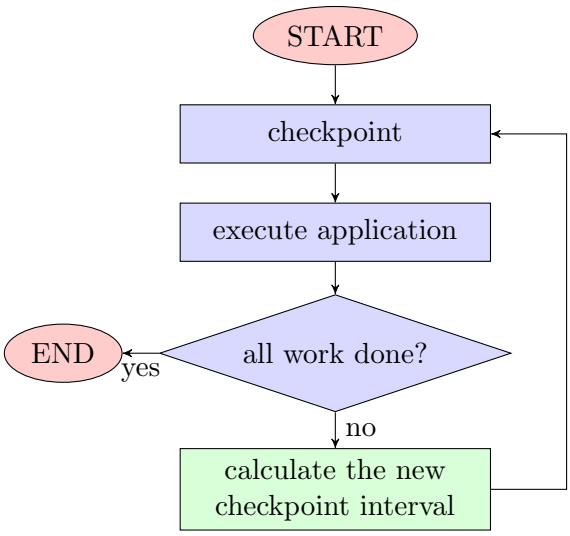


Figure 3.8: Diagram depicting the dynamic re-definition of the checkpoint interval.

This methodology is based on measurements taken during the most recently checkpoint cycle. When the application changes its behaviour, *i.e.* the communication pattern or its memory footprint, after one checkpoint cycle the checkpoint interval will already be adapted to the new application characteristics.

Moreover, except during the start-up and finalisation phases it is expected that applications do not change their behaviour or memory footprint too frequently in comparison to the checkpoint interval (80).

Of what worth are convictions
that bring not suffering?
— **Antoine de Saint-Exupéry**

4

Experimental Evaluation

This chapter presents the experimental evaluation of the proposed checkpoint interval model. First of all the experimental environment is described. The description of the experimental environment includes the fault distribution used to run the experiments, the event-based simulator used to run simulation-based experiments, and the fault tolerant MPI library.

The experimental section is divided in two subsections. The first subsection presents a comparison of the proposed model for coordinated checkpointing with other author's models. The second subsection presents experiments related to the uncoordinated checkpointing model, including the inter-process dependency factor.

To close this chapter, the experiments conclusions are presented.

4.1 Experimental Environment

To run experiments a 32 node cluster has been used. Each node is equipped with two Dual-Core Intel Xeon processors running at 2.66GHz, performing a total of 128 cores on the parallel machine. Moreover, each node has 12 GBytes of main memory and a 160 GByte SATA disk for local storage. Nodes are interconnected via two Gigabit Ethernet interfaces. One of these networks is used for storage and event logging while the other is used for process communication.

To inject faults a program has been designed. This program runs on a machine external to the cluster. According to the fault distribution, the program connects to the target node and kills the application process. The target machine is selected in a round-robin fashion. After the process kill, the node becomes available to recover the killed process. Nodes are not overloaded with more than one process per core.

4.1.1 Fault Distribution

In order to compare models a fault distribution should be used. This subsection describes the fault distribution used for simulation and real executions in this paper.

The fault distribution has been created based on the MT19937 pseudo-random numbers generator algorithm (60). We have used the first 10 thousands prime numbers as seeds for this generator. All these seeds generate pseudo-random numbers that are normalised to a range from -100 to 100. The sum of these numbers converges to zero after 500 rounds, with less than 2.5% of relative error, as presented in figure 4.1. After 4000 rounds the relative error is smaller than 1%.

After verifying that values taken using the MT19937 algorithm converge to a medium, these values can be safely used in our experiments. This verification is necessary because the models rely on the mean time

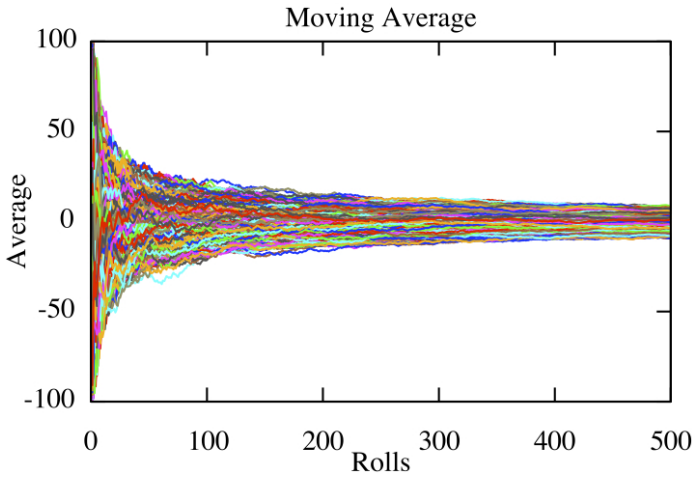


Figure 4.1: Moving average of values generated by 10 thousand different seeds after 500 rounds. Graph shows the convergence to an average value with less than 2.5% of relative error. These seeds have been used for simulation-based experiments.

to interrupt (MTTI) value for a given computer. Thus, our fault distribution must assure that the median value exists after a specific number of faults. In this paper, experiments run long enough to present at least 500 faults.

The fault distribution is defined based on the MTTI value specified for each experiment and pseudo-random numbers series described above. For each MTTI value the normalised correspondent pseudo-random number is added. This means that for the {-67, 43, -24, 74, -72} pattern and with 100 minutes as a value for MTTI, faults will be placed on minutes 33, 143, 176, 328 and 374, as shown in figure 4.2. It means that the 4th fault occurs after the 5th fault. There is the possibility of up to 3 faults occurring simultaneously.

As aforementioned, experiments should run for long enough for at least 500 faults to appear. This means that for a given MTTI value, experiments run for at least 500×MTTI long. Simulations run for all 10 thousands seeds. However, experiments using real applications run for

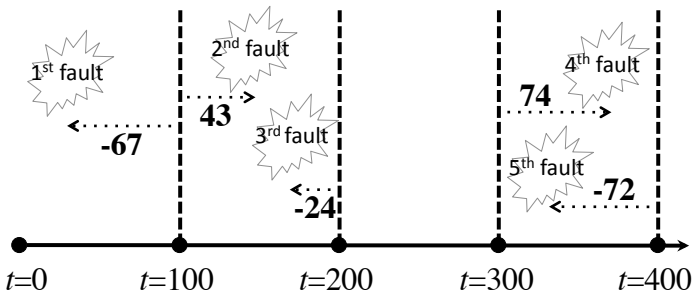


Figure 4.2: The fault distribution based on a given MTTI (100) and pseudo-random numbers generated by MT19937 algorithm (-67, 43, -24, 74, -72, ...).

only a few selected seeds due to the time needed to perform all those experiments presented in this section. For this reason, a total of 21 seeds have been selected.

All these seeds converge close to the 500th round, with a relative error smaller than 0.5% as shown in figure 4.3. These seeds have been selected due to their convergence moment and the small relative error presented

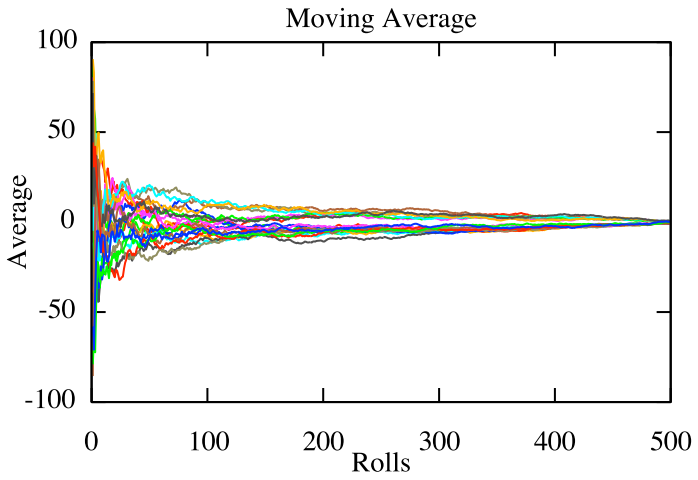


Figure 4.3: Moving average of values generated by 21 selected seeds after 500 rounds. Graph shows the convergence to an average value with less than 0.5% of relative error close to the 500th round. These seeds have been used for real execution based experiments.

Table 4.1: Seeds selected to generate fault distribution for real execution experiments.

Seed	Rounds to converge	Seed	Rounds to converge
6991	492	16693	494
21341	495	22153	494
25603	493	26783	491
36929	492	42239	492
51517	495	52967	494
54443	493	59729	492
64157	493	69401	492
70981	494	71537	494
76543	493	88867	494
95219	493	100609	495
103483	491		

by the final convergence value. Table 4.1 presents selected seeds and the number of rounds necessary to converge.

With the fault distribution described above it is possible to recreate all the experiments presented in this paper. Other fault distributions can be used, mainly to analyse the overhead generated according to a specific scenario. However, in order to compare models we have chosen a neutral fault distribution that disperses faults uniformly over a wide time range.

4.1.2 Fault Tolerant MPI Library

RADIC/OMPI (42) has been used as a fault tolerant MPI library. According to the RADIC architecture (33), on each node there are three entities: protector, observer, and the application process (P_n), as figure 4.4 depicts. Observers manage MPI communication, perform message logging and take checkpoints. A protector located on another node stores these data. Protectors use a heartbeat/watchdog mechanism to detect faults. Fault tolerance communication, including checkpointing, message logging, and heartbeat runs over the storage network.

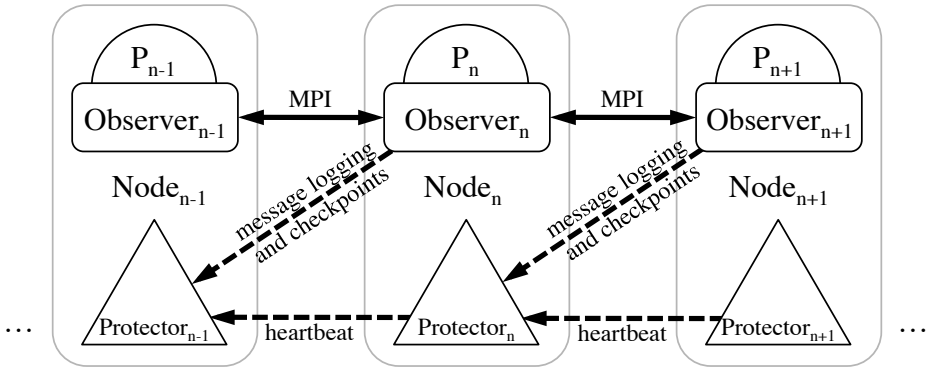


Figure 4.4: Architecture of the RADIC/OMPI fault tolerance library. Dashed lines are fault tolerance specific communication and continuous lines are MPI communication.

The RADIC/OMPI library performs uncoordinated checkpointing combined with pessimist receiver-based message logging. There is no central element on the RADIC architecture, neither for the fault tolerance operation nor storage (33, 42). RADIC/OMPI uses storage resources available on nodes in order to create a distributed stable storage.

4.2 Experiments

4.2.1 Model for Coordinated Checkpointing

Hereunder, a comparison between the model designed for serial applications or coordinated checkpoint and previous models will be presented. The comparison was made using simulation and real executions.

4.2.1.1 Comparison Using Real Executions

In order to compare models using real executions, a synthetic program that creates the experimental environment has been designed. Figure 4.5 shows the architectural diagram of the system used. It consists of a main process composed of 3 threads and one separated process.

The threads of the main process are: fault injector, checkpointer, recoverer, and the target application is the separated process. As the system starts, both processes are launched simultaneously.

The checkpointer thread immediately takes a checkpoint of the target application process and measures the time needed to perform this action. With this information it calculates the next checkpoint interval. The checkpoint interval is recalculated after each checkpoint in order to reflect changes in the application memory footprint.

The fault injector thread kills the target application process according to the fault distribution in use. The recoverer thread monitors the target application process to detect faults according to the maximum detection latency used. When a fault is detected, the recoverer thread restores the target application process from the last checkpoint. Thus, the target application resumes its execution until it reaches the end.

As the target application we have used a simple matrix multiplication algorithm. To change the input value of the models variables different

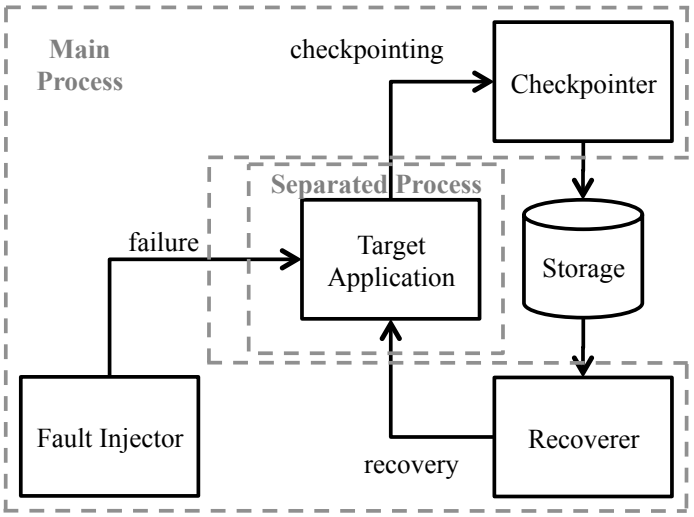


Figure 4.5: Architecture of the system used for model comparison using real applications.

matrix sizes have been used. It is expected that the overhead increase according to matrix dimensions. Nevertheless, the overhead prediction error should stay stable and delimited by a small relative error range.

For this experiment the fault detection latency has been set to zero, the MTTI has been defined to be 100 seconds, and all 21 fault distributions generated by those seeds shown in table 4.1 have been used.

The checkpoint interval is calculated after each checkpoint, using the most recent measurements of the time needed to perform a checkpoint. Table 4.2 presents the checkpoint sizes and discloses the mean time needed to take (t_c) and load (t_l) a checkpoint. Each experiment has been executed at least 16 times and values depicted in graphs are the average of all data that fall in a 95% confidence interval.

Table 4.2: Checkpoint sizes according to matrix dimensions. The last two columns show the mean time needed to perform a checkpoint (t_c) and the mean time needed to load it from storage (t_l).

Matrix Dimension	**Checkpoint Size (MB)**	t_c **(seconds)**	t_l **(seconds)**
200×200	1.138	0.034	0.025
300×300	2.274	0.042	0.052
400×400	3.880	0.062	0.077
500×500	5.938	0.099	0.111
600×600	8.462	0.143	0.143
700×700	11.438	0.182	0.188
800×800	14.868	0.247	0.236
900×900	18.763	0.303	0.296
1000×1000	23.106	0.386	0.362
1100×1100	27.915	0.451	0.425
1200×1200	33.177	0.532	0.517
1300×1300	38.892	0.627	0.587
1400×1400	45.083	0.719	0.680
1500×1500	51.716	0.877	0.820
1600×1600	58.813	0.948	0.878
1700×1700	66.360	1.086	0.998

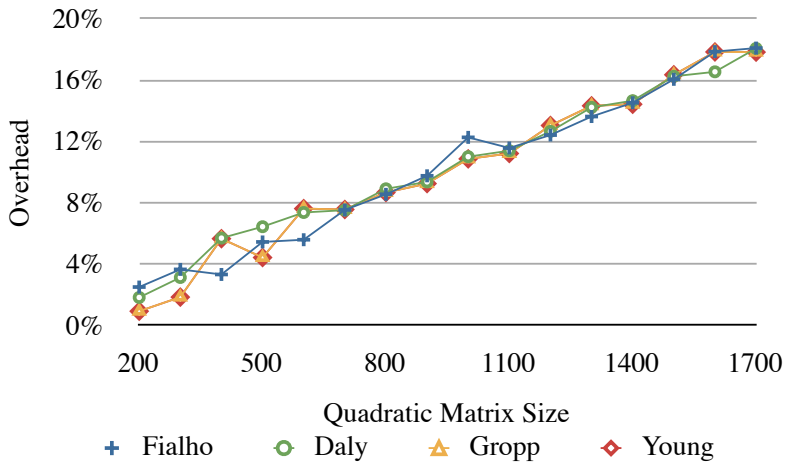


Figure 4.6: Overhead introduced by fault tolerance on the application run time. Model performance is close in all cases. Values are measurements, not predictions.

Figure 4.6 shows that all models introduce practically the same overhead on the application run time for a memory size greater than 10 megabytes, represented by a 700×700 matrix size. For checkpoint files smaller than 10 megabytes, values are scattered. This is an expected behaviour, since the time needed to checkpoint processes with small memory footprint is more sensitive to operating systems and hardware perturbations.

Analysing figure 4.6 it is possible to perceive that the differences in checkpoint interval models do not affect the fault tolerance overhead in a perceptible way. On the other hand, model prediction for the application run time loses accuracy when the checkpoint size grows. Predictions have been calculated using overhead equations shown in table 3.2.

Figure 4.7 presents the model prediction accuracy. Daly's model is more precise than any other. Its prediction error is smaller than that of the others. Nevertheless, all models present a relative prediction error smaller than 2.5% for workloads smaller than 1500×1500.

For matrix dimensions greater than 1500×1500 the relative error in-

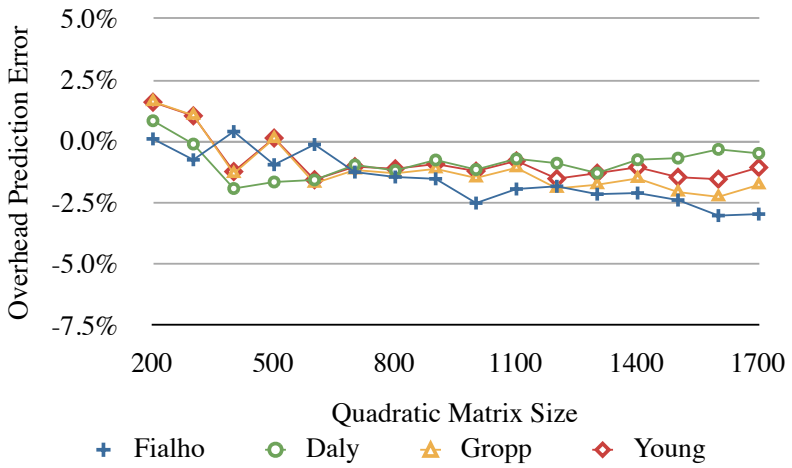


Figure 4.7: Relative prediction error presented by models in comparison with the real execution time.

creases for all models, with the exception of Daly's model which presents a relative error smaller than 4% until a 4000×4000 matrix size.

This next experiment tries to verify the accuracy of the calculated checkpoint interval to minimise the overhead introduced by fault tolerance. The predicted overhead relative error is analysed also. For that, the synthetic program has been executed with different checkpoint intervals. The application overhead is compared with the predicted overhead of the models.

For this experiment a medium-size matrix (1200×1200) has been chosen. This matrix size has been selected because it is not too small to be affected by operating system or hardware perturbations. The fault detection latency has been set to zero, the MTTI has been defined to be 100 seconds, and all 21 fault distributions generated by those seeds shown in table 4.1 have been used. For each checkpoint interval the application has been executed at least 16 times and values depicted in graphs are the average of all data that fall in a 95% confidence interval.

Figure 4.8 depicts a comparison between overhead prediction of the models and a real execution. All models have calculated an optimum

Table 4.3: Comparison of a real execution and model prediction. Lines show the relative error of predicted overhead of the models for each checkpoint interval used. The last line presents the optimum checkpoint interval estimation of the models.

Interval	Fialho	Daly	Young	Gropp
7	2.6%	1.2%	1.5%	1.8%
8	2.1%	0.8%	1.2%	1.5%
9	1.8%	0.6%	1.0%	1.3%
10	1.8%	0.6%	1.0%	1.3%
11	1.7%	0.5%	0.9%	1.2%
12	1.9%	0.7%	1.0%	1.5%
13	2.1%	0.8%	1.2%	1.7%
Predicted	9.752s	9.766s	10.300s	10.300s

checkpoint interval between 9.75 and 10.3 seconds, as shown in table 4.3. According to the real execution, the optimum checkpointing interval seams to be between 10 and 11 seconds. All models have presented an overhead relative error smaller than 3% as shown in table 4.3. Close to the optimum checkpoint interval calculated by the models, the relative

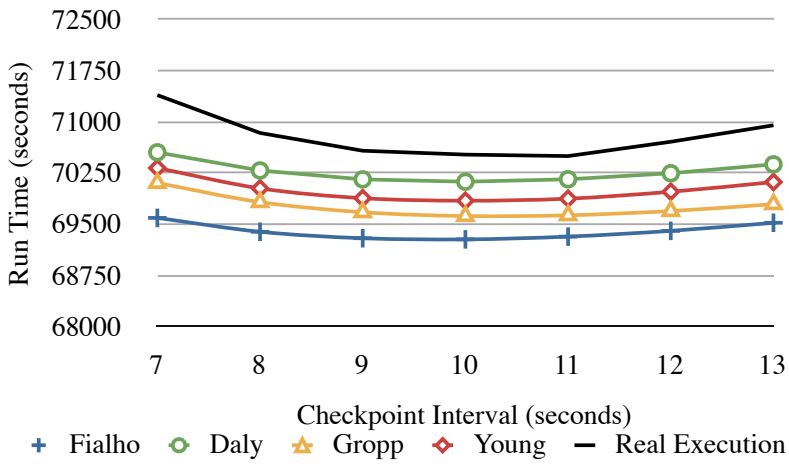


Figure 4.8: Comparison of real execution and overhead prediction of the models for $\alpha = 100$, $t_c = 0.530$, $t_l = 0.505$, $t_d = 0$, values in average. Application runs in 62,830 seconds without fault tolerance and in absence of faults.

error is smaller than 2% for all models.

4.2.1.2 Comparison Using Simulation

In order to compare models through simulation, a discrete event simulator has been designed. Figure 4.9 shows a flow diagram depicting consecutive events on this simulator. To place a fault at the right moment there is a conditional before each event. As input, the simulator receives the fault distribution, the MTTI, the time needed to take and load a checkpoint, and the application run time. The output is the simulated run time for each checkpoint interval starting from 1 to MTTI.

This simulator has been used to analyse the relation between MTTI and the time needed to take and load a checkpoint. This simulation shows the MTTI impact on model accuracy, fixing all variables other than MTTI and the checkpoint interval. For that, an extremely long running application (500 days) will be simulated to assure a minimum of 500 faults in each simulation.

Table 4.4 shows variables used for simulation, the detection latency has been set to zero, and all fault distributions generated by the aforementioned 10 thousands seeds have been used. Moreover, table 4.4 shows the values of the optimum checkpoint interval calculated by models for each scenario.

Table 4.4: Variables used to simulate the influence of MTTI on model accuracy and the optimum checkpoint interval calculated by each model.

MTTI	**Run Time**	t_c	t_l	**Young**	**Gropp**	**Daly**	**Fialho**
hours	days	minutes	minutes	minutes	minutes	minutes	minutes
24	500	5	5	120.00	120.00	115.00	114.89
6	500	5	5	60.00	60.00	55.00	54.79
1	500	5	5	24.49	24.49	19.49	18.98

Figure 4.10 shows a comparison of the simulation and all four models

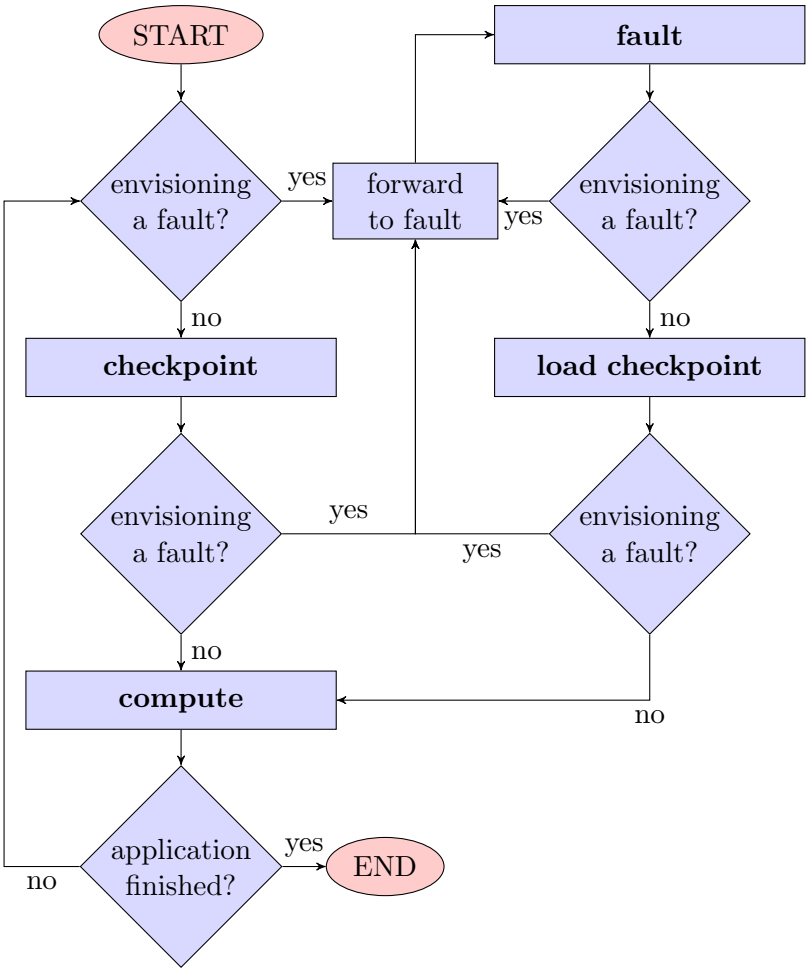


Figure 4.9: Diagram of the discrete event simulator used. Each box represents an event, and its time is added to the simulated run time.

using a 24 hour MTTI. A 10 minute checkpoint interval step has been used between each simulated checkpoint interval. The results of the models are very close to the simulation.

With regard to the predicted overhead, the proposed model presents a relative error of 0.64% on values close to the calculated checkpoint interval (114.89 minutes). Daly's model presented an error smaller than 0.2% for

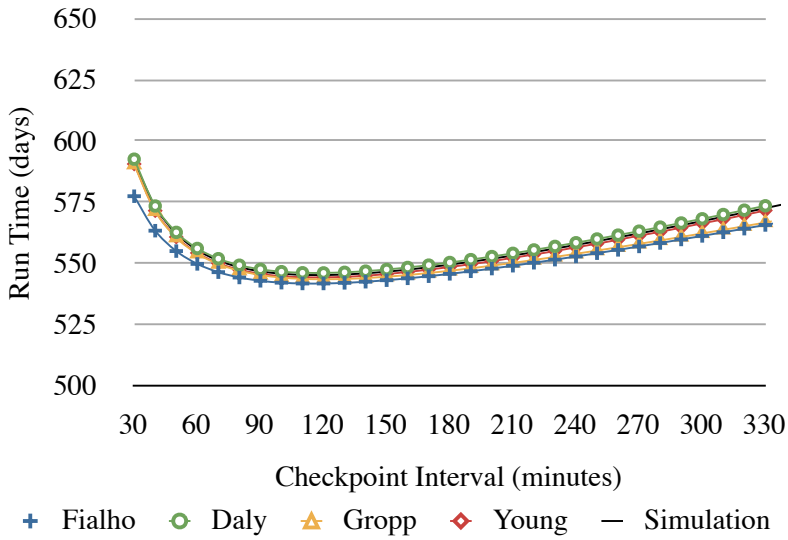


Figure 4.10: Comparison of models and simulation results for values depicted in table 4.4 using a 24 hours MTTI. The minimum overhead is achieved for a checkpoint interval value between 110 and 130 minutes.

all checkpoint intervals simulated. As expected, all models perform well when the ratio between the time needed to take a checkpoint is relatively small in comparison to MTTI.

Figure 4.11 depicts the same experiment but using a 6 hours MTTI. It demonstrates that Young's model cannot predict the overhead with less than 15% error under these circumstances. However, its calculated checkpoint interval is still close to the optimum. Other models also start to show an increase in the overhead prediction error. The smaller simulated run time is for a 55 minute checkpoint interval. At this point, the proposed model, Gropp's and Daly's present a relative error of 2.54%, 1.23% and 0.67%, respectively. In this experiments a 5 minute checkpoint interval step has been between each simulated checkpoint interval.

As the MTTI decreases, Young's model is completely unable to predict the overhead as shown in figure 4.12. Gropp's model predicts the overhead better than the proposed model, which presents an error of 16%. In this

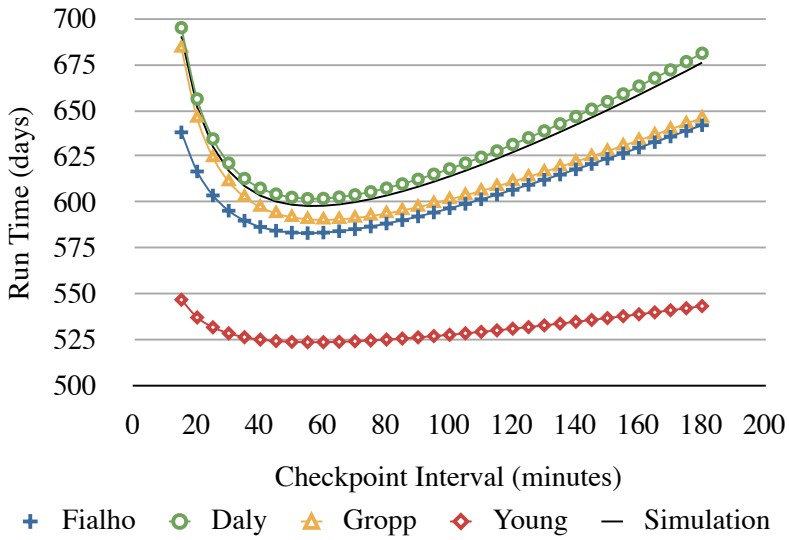


Figure 4.11: Comparison of models and simulation results for values depicted in table 4.4 using a 6 hours MTTI. The minimum overhead is achieved for a checkpoint interval value between 50 and 60 minutes.

experiment a one-minute checkpoint interval step has been used between simulations. The smaller overhead present in the simulation has been achieved for a 21 minute checkpoint interval. This value is very close to the values calculated by Daly's and the proposed models.

4.2.2 Model for Uncoordinated Checkpointing

The experimental evaluation for the uncoordinated checkpointing presented in this section is divided in three set of experiments. The first set presents the comparison of the proposed model with other author's models. The second set of experiments concerns the accuracy and effectiveness of the checkpoint interval model and of the inter-process dependency factor. The third and last set of experiments is related to the adaptation of the checkpoint interval to changes on the application's characteristics. Additionally, in the second and third set of experiments other author's models are also presented in comparison with the proposed models.

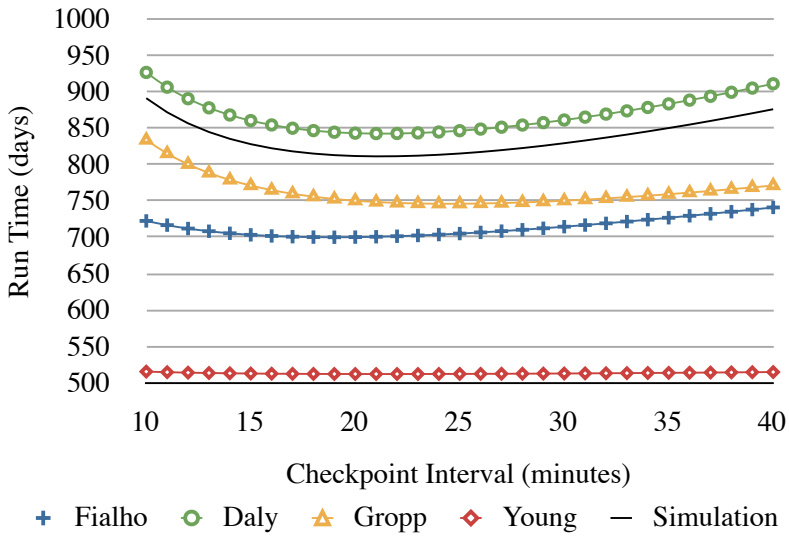


Figure 4.12: Comparison of models and simulation results for values depicted in table 4.4 using a 1-hour MTTI. The minimum overhead is achieved for a checkpoint interval value between 20 and 22 minutes.

Other models had been designed to be used with serial applications. To compare these models with the proposed one may be unfair while running applications protected by uncoordinated checkpoint combined with message logging. However, it is important to show de benefits of using models specifically designed for uncoordinated checkpointing.

4.2.2.1 Comparison With Other Models

In order to compare the proposed model with other models a set of experiments has been designed. These experiments use the NAS LU application running with 8 processes, one *per* node. This application has been executed using class B and C. Table 4.5 depicts relevant characteristics of both applications.

Δ values reflect the average measurements done during application execution. In order to achieve a minimum execution time of 50,000 seconds, the number of iterations of LU class B and C has been modified

Table 4.5: Summary of relevant characteristics of NAS LU application. Time values are expressed in seconds. The rightmost column depicts the optimum checkpoint interval in seconds calculated by our model. $t_d = 0.5$, $\alpha = 100$, $\phi = 1$.

LU Class	t_c	t_l	Δ_{lp}	Δ_{lr}	σ
B	0.605	0.559	38.257	0.005	**10.353**
C	2.057	2.102	13.961	0.007	**18.065**

to 300,000 and 37,500 respectively. As shown in table 4.5, due to the receiver-based message logging protocol used by RADIC/OMPI the Δ_{lr} value is virtually zero.

Table 4.6 shows a summary of measurements and overhead predictions for the NAS LU. Percentiles refer to the predicted overhead related to the original run time.

As shown, other models are not useful to predict the overhead for applications where message logging affects its performance. As figure 4.13 depicts, for those kinds of applications the proposed model performs better than any other model with regard to the estimated execution time. This occurs because the message logging increases the applications execution time and other models do not consider this interference.

However, all models calculate an accurate checkpoint interval. The proposed model estimates a 10.353 seconds checkpoint interval. Young's and Gropp's models estimate an 11 seconds checkpoint interval while Daly's a 10.395, as shown in table 4.8. As figure 4.13 shows, the optimum checkpoint interval is probably between 10 and 12 seconds. The accuracy of the checkpoint interval calculation presented by other models is expected because the value of ϕ is 1 for this application.

Table 4.7 presents a summary of models overhead prediction relative error for the application execution presented above. As shown, our model has presented an error smaller than or close to 4% for all these applications.

Table 4.6: Summary of measurements and predictions for the NAS LU application. Time values are expressed in seconds. Percentiles refer to the predicted overhead with relation to the original run time. $t_d = 0.5$, $\alpha = 100$, $\phi = 1$, $\sigma = 10$ for LU class B, and $\sigma = 18$ for LU class C. Other variable values are available in table 4.5.

	Execution Measures			Fialho		Daly		Gropp		Young	
Application	Un-Protected Run Time	Protected Execution Time		Predicted Run Time		Predicted Run Time		Predicted Run Time		Predicted Run Time	
LU class B	68,469	107,374	36.2%	102,484	33.2%	77,030	11.1%	76,418	10.4%	79,868	14.3%
LU class C	36,093	48,948	26.3%	48,594	25.7%	45,478	20.6%	44,225	18.4%	45,371	20.4%

Table 4.7: Summary of run time prediction relative error for NAS LU class B running with 8 processes. Our model presents the smallest error in all cases. Other variables are available in table 4.5. Checkpoint interval values are represented in seconds.

Interval	Fialho	Daly	Gropp	Young
7	3.35%	24.96%	25.59%	21.56%
8	3.24%	25.10%	25.76%	21.82%
9	3.12%	25.13%	25.81%	21.93%
10	3.06%	25.14%	25.85%	21.97%
11	3.12%	25.21%	25.96%	22.04%
12	3.24%	25.31%	26.08%	22.09%
13	3.36%	25.36%	26.18%	22.09%
14	3.41%	25.34%	26.18%	22.01%

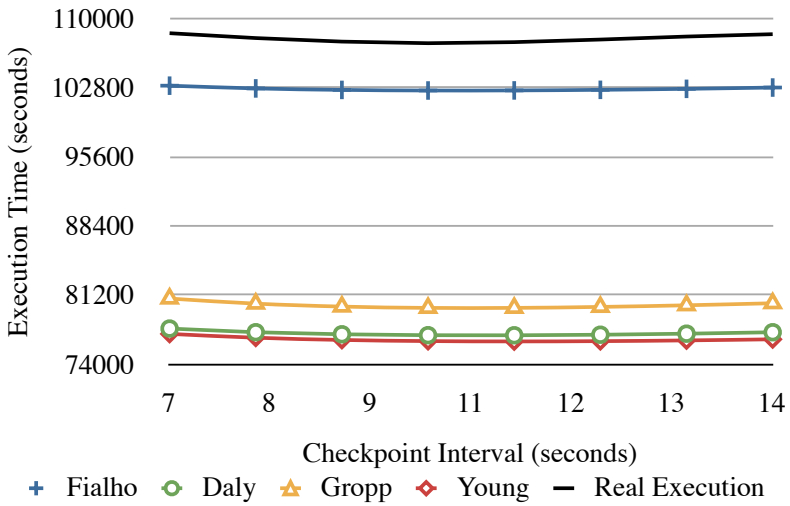


Figure 4.13: Comparison of model overhead prediction and real execution of LU class B. Values of variables are depicted in table 4.5, $\alpha = 100$, and $t_d = 0.5$. Values are expressed in seconds. The minimum overhead is achieved for a checkpoint interval value between 10 and 12 seconds.

Table 4.8: Estimated checkpoint interval for NAS LU class D. Values of variables are available in table 4.5, $\alpha = 100$ and $t_d = 0.5$. Values are expressed in seconds.

Application	Fialho	Daly	Gropp	Young
LU class B	10.353	10.395	11.000	11.000

4.2.2.2 Accuracy and Effectiveness of the Model

To verify the accuracy of the proposed model the NAS (77) LU class B with 8 processes modified to iterate 300,000 times has been executed. This modified version of the LU has been executed using different global values for the ϕ, from 0.9 to 0.2. Table 4.9 shows the correct value of this factor individually and globally for this application.

Analysing the curve in figure 4.14 and the data present in the table 4.15 it is possible to guess that the optimum value for the ϕ for this execution should be a value between 0.45 and 0.60. Despite of the small difference between the global and the individualised values for the ϕ, to

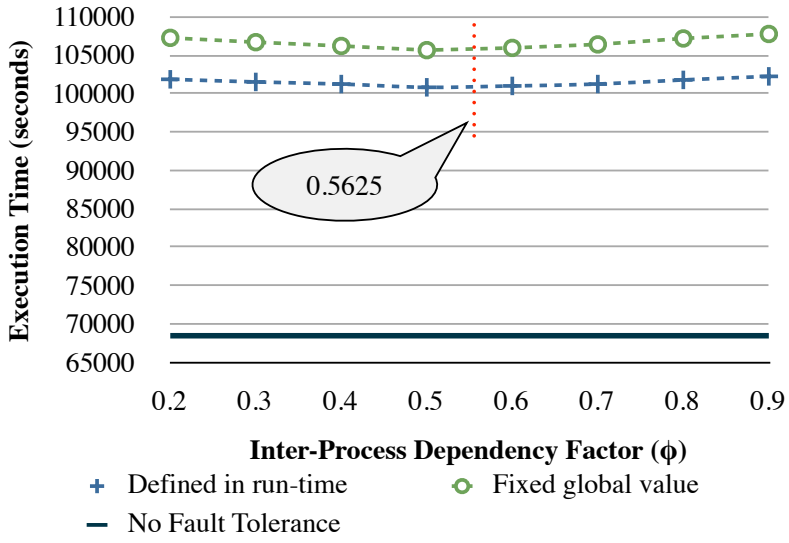


Figure 4.14: Comparison between static and in run-time configured values of the inter-process dependency factor.

use of a precise value for this factor reduces in more than 3% the overhead introduced by the fault tolerance tasks for this application.

To evaluate the effectiveness of the checkpoint interval model for parallel application two sets of experiments have been designed: *1)* one analyses the effectiveness of the inter-process dependency factor, and *2)* verifies the correctness of the message logging modelling. To show that the using of current models is inappropriate in this scenario, the values achieved with other models will be included on the following experiments.

To run experiments presented in this section the MTTI (α) has been set to 100 seconds. The RADIC/OMPI library has been configured to

Table 4.9: Values for the inter-process dependency factor for the entire LU application and for each process individually.

Process Rank *(Global Value)*	$P(n)$	ϕ
Running with 8 processes		**0.56250**
0, 3, 4, 7	4	0.50000
1, 2, 5, 6	5	0.62500

Figure 4.15: Values for the overhead introduce by fault tolerance tasks on the execution of the NAS LU application according to the configuration methodology.

ϕ	**Overhead Using Fixed Global Value**	**Overhead Using a Defined in Run-Time Value**	**Difference**
0.2	36.2%	32.8%	3.38%
0.3	35.8%	32.6%	3.27%
0.4	35.5%	32.4%	3.16%
0.5	35.2%	32.1%	3.11%
0.6	35.4%	32.2%	3.17%
0.7	35.7%	32.4%	3.28%
0.8	36.1%	32.7%	3.39%
0.9	36.5%	33.0%	3.43%

send a heartbeat every 1 second. Thus, the fault detection latency (t_d) is 0.5 seconds. As this library performs receiver-base message logging during the recovery phase messages are already available in the log. Thus, the time needed to process the message log (Δ_{lr}) tends to be unappreciable because there is no message replaying (72).

To assure that the inter-process dependency factor is the only variable which changes between executions a synthetic application has been designed. The message logging interference and the time needed to take and load a checkpoint are quite similar for the same number of process *per* node.

The synthetic application has been programmed using the SPMD paradigm. Figure 4.16 presents the execution flow of each process of the synthetic application. A computing and a communication phase compose each process. The computing phase is represented by a 2000×2000 matrix multiplication and during the communication phase processes communicate to the right and lower neighbours, as depicted in figure 4.17.

Table 4.10: Relevant characteristics of the synthetic application used to verify the effectiveness of the inter-process dependency factor. For each model, the first column shows the optimum checkpoint interval calculated and the second column shows the predicted overhead error. $\alpha = 100$ and $t_d = 0.5$. Values are expressed in seconds.

# of						Fialho		Daly		Gropp		Young	
Processes	ϕ	t_c	t_l	Δ_{lp}	Δ_{lr}	σ	**Error**	σ	**Error**	σ	**Error**	σ	**Error**
4	1.0000	1.630	1.643	2.771	0.000	16.30	3.67%	16.42	2.8%	18.05	4.7%	18.05	3.4%
9	0.5556	1.622	1.596	2.763	0.000	22.39	3.15%	16.39	1.3%	18.01	0.6%	18.01	0.0%
16	0.3125	1.691	1.610	2.765	0.000	31.00	2.21%	16.70	5.3%	18.39	3.3%	18.39	3.3%
25	0.2000	1.650	1.634	2.779	0.000	38.70	2.11%	16.52	7.0%	18.17	5.1%	18.17	4.8%
16 (4×4)	0.3125	4.954	5.131	4.188	0.000	50.46	3.08%	26.52	12.1%	31.48	7.0%	31.48	6.4%
36 (4×9)	0.1389	5.032	5.199	4.160	0.000	78.73	2.65%	26.69	17.4%	31.72	12.2%	31.72	10.4%
64 (4×16)	0.0781	4.981	5.287	4.399	0.000	106.06	1.88%	26.58	20.4%	31.56	15.3%	31.56	12.7%
100 (4×25)	0.0500	5.284	5.330	4.328	0.000	137.76	1.63%	27.22	22.9%	32.51	17.6%	32.51	14.6%

Table 4.11: Characteristics of the NAS LU class B and C. For each model, the first column shows the optimum checkpoint interval and the second the predicted overhead error. $\alpha = 100$, $t_d = 0.5$, $\phi = 0.5625$. Values are expressed in seconds.

LU					Fialho		Daly		Gropp		Young	
Class	t_c	t_l	Δ_{lp}	Δ_{lr}	σ	**Error**	σ	**Error**	σ	**Error**	σ	**Error**
B	0.605	0.559	38.257	0.005	10.353	3.0%	10.395	25.1%	11.000	25.9%	11.000	22.0%
C	2.057	2.102	13.961	0.007	18.065	0.5%	18.065	5.6%	20.283	8.0%	20.283	5.8%

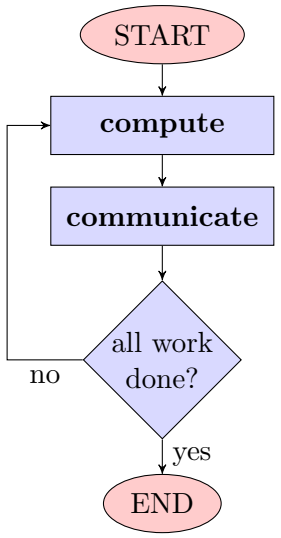


Figure 4.16: Execution flow of each process of the synthetic application used to verify the effectiveness of the inter-process dependency factor.

These phases are repeated until a defined amount of work has been done. The computing and communication load are the same for all executions. Thus, the interference caused by the message logging is the same

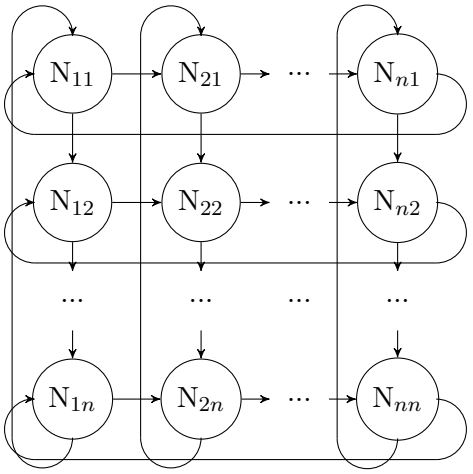


Figure 4.17: Communication patter of the synthetic application used to verify the effectiveness of the inter-process dependency factor

in all experiments regardless of the number of processes used. However, the value of the inter-process dependency factor changes accordingly to the number of processes.

Besides other variables, table 4.10 depicts the value of ϕ for all executions. Values of message logging operation (Δ_{lp} and Δ_{lr}), checkpoint taking (t_c), and checkpoint loading (t_l) are averages of all measurements done during application execution. The value of the checkpoint interval has been previously calculated based on the applications characteristics and is used to configure the RADIC/OMPI library.

As shown in figures 4.18 and 4.19, as the number of processes increases (or the value of ϕ decreases) so does the accuracy of our model. On the execution with 4 processes the value of ϕ is 1. In this case all models perform similarly. However, as the number of processes increases other models depicts a loss of accuracy.

Analysing figure 4.19 it is easy to conclude that previous models cannot be used with parallel applications protected by uncoordinated check-

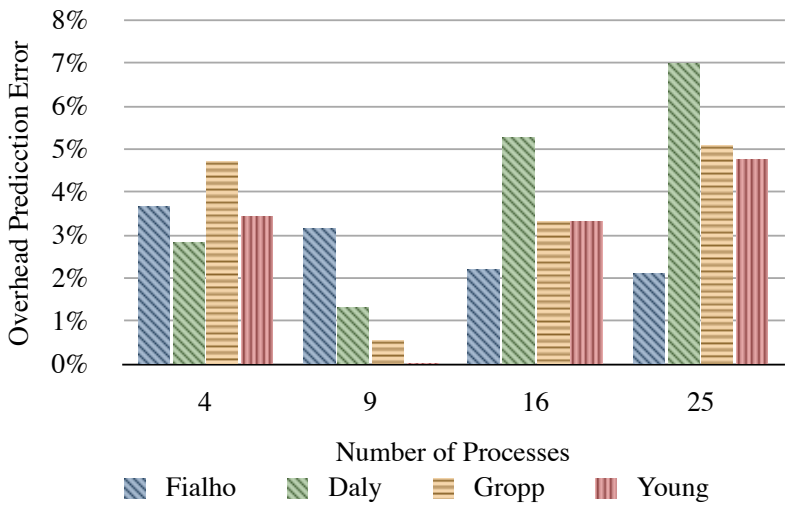


Figure 4.18: Overhead prediction error for a synthetic application running with 4, 9, 16, and 25 processes, 1 *per* node. Values of variables are depicted in table 4.10, $\alpha = 100$, and $t_d = 0.5$.

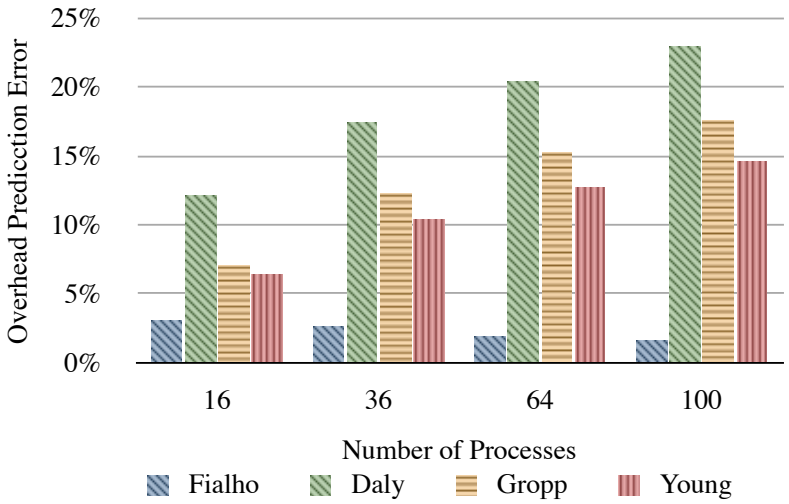


Figure 4.19: Overhead prediction error for a synthetic application running with 16, 36, 64, and 100 processes, 4 *per* node. Values of variables are depicted in table 4.10, $\alpha = 100$, and $t_d = 0.5$.

points combined with message logging. Especially with a high number of processes.

In order to verify the correctness of the message logging modelling a new set of experiments have been made. These experiments use the LU application from the NAS Parallel Benchmarks (77) running with 8 processes, one *per* node. LU has been executed using class B and C.

Table 4.5 depicts relevant characteristics of LU. Δ values reflect the average measurements done during application execution. The number of iterations of LU class B and C has been modified to 300,000 and 37,500 respectively. The LU application presents a ϕ value equal to 0.5625 for 8 processes.

As shown in figures 4.20 and 4.21 the proposed model performs better than any other. Because other models do not consider the message logging time they present an overhead prediction relative error greater than 20% for LU class B. It means that these models are not useful to predict the overhead for parallel applications using uncoordinated checkpointing

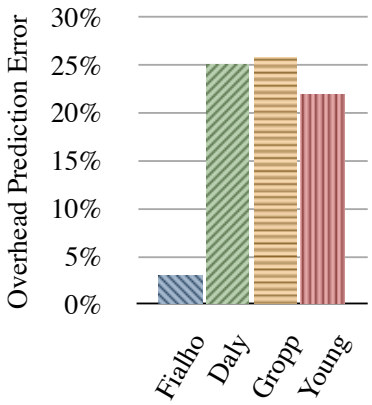


Figure 4.20: Model overhead prediction relative error for LU class B. Values of variables are depicted in table 4.5.

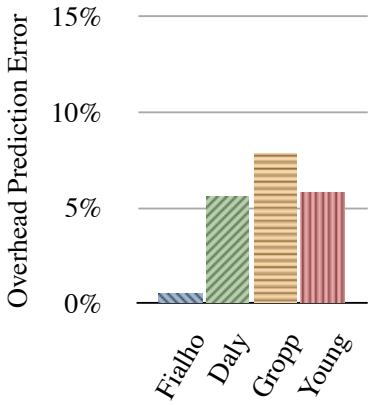


Figure 4.21: Model overhead prediction relative error for LU class C. Values of variables are depicted in table 4.5.

combined with pessimist receiver-based message logging.

Nevertheless, the proposed model model presents a modest overhead prediction error for both class B and C of the NAS LU. Notice that from figure 4.20 to figure 4.21 other models presented a decrease in the overhead prediction error while the opposite occurs with our model. This occurs because the ratio between compute and communication changes reducing the interference of message logging.

The next experiment tries to depict the sensibility of the inter-process

dependency factor to the application's communication pattern and number of processes. For that four applications from the NAS Parallel Benchmarks (77) have been chosen. The applications are: CG, LU, BT and SP.

Table 4.12 shows the values for the inter-process dependency factor (ϕ) for the selected applications. As expected, different communication patterns generates different values for the ϕ. Moreover, in these applications the number of peers each process communicate is defined by the data structure used by applications, and the applications is programmed under an SPMD-like paradigm. This means that the number of peers each process communicate does not increases with the same ration with which the number of processes increases.

Figure 4.22 depicts the same data presented in table 4.12. As we can see, for CG and LU applications the value of the ϕ decreases as the number of processes increases. However, SP and BT present a special decrease in the number of peers each process communicate to when the total number

Table 4.12: Values for the inter-process dependency factor for CG, LU, BT, and SP applications from the NAS suite according to the number of processes used to run the application.

# of		ϕ	
Processes	CG	LU	BT/SP
4	1.00000	0.75000	1.00000
8	0.62500	0.56250	—
9	—	—	1.00000
16	0.37500	0.37500	0.56250
25	—	—	0.49920
32	0.21875	0.22656	—
36	—	—	0.40895
49	—	—	0.34402
64	0.12500	0.13281	0.17188
81	—	—	0.20668
100	—	—	0.18870
121	—	—	0.15272
128	0.07031	0.07520	—

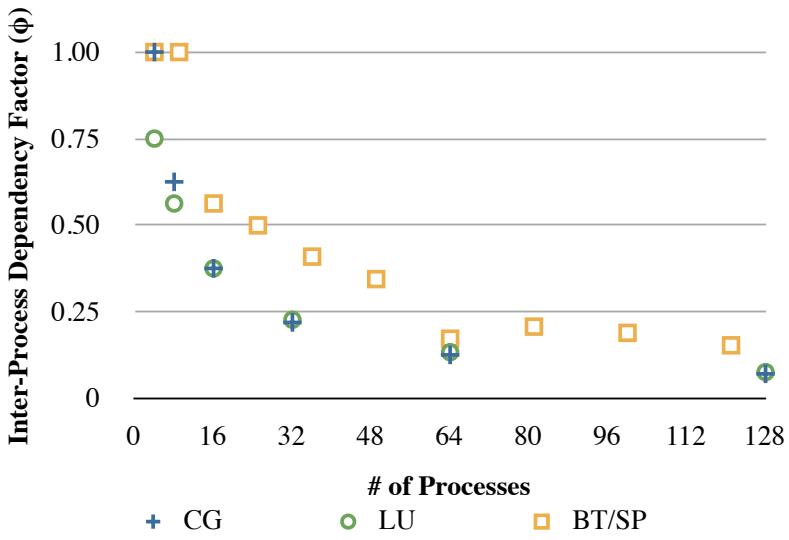


Figure 4.22: Inter-process dependency factor for CG, LU, BT, and SP applications from the NAS suite according to the number of processes used to run the application.

of processes is a perfect cube (64). These applications present the same communication pattern and data-mapping algorithm.

To depict the influence of inter-process dependency factor on the overhead prediction we chose the LU application from the NAS suite. This application has been chosen due to its good scalability. Table 4.13 depicts the global value for this factor for the LU applications according to the number of processes it used to run. Also, the value of the inter-process dependency factor for each individual process is exhibited.

Figure 4.23 depicts both the overhead prediction error and the inter-process dependency factor for the LU applications according to the number of processes. As the value of the ϕ decreases the overhead prediction error of our checkpoint interval model starts decreasing for a small number of nodes and stabilises after 16 nodes in use.

The overhead prediction error is the difference between the overhead calculated by the cost function introduced in equation 3.19 and the over-

Table 4.13: Values for the inter-process dependency factor for the entire LU application and for each process individually.

Process Rank *(Global Value)*	$P(n)$	ϕ
Running with 4 processes		**0.75000**
0–3	3	0.75000
Running with 8 processes		**0.56250**
0, 3, 4, 7	4	0.50000
1, 2, 5, 6	5	0.62500
Running with 16 processes		**0.37500**
0, 3, 12, 15	5	0.31250
1, 2, 4, 7, 8, 11, 13, 14	6	0.37500
5, 6, 9, 10	7	0.43750
Running with 32 processes		**0.22656**
0, 7, 24, 31	6	0.18750
1–6, 8, 15, 16, 23, 25–30	7	0.21875
9–14, 17–22	8	0.25000
Running with 64 processes		**0.13281**
0, 7, 56, 63	7	0.10938
1–6, 8, 15, 16, 23, 24, 31, 32, 39, 40, 47, 48, 55, 57–62	8	0.12500
9–14, 17–22, 25–30, 22–28, 41–46, 49–54	9	0.14063
Running with 128 processes		**0.07520**
0, 15, 112, 127	8	0.06250
1–14, 16, 31, 32, 47, 48, 63, 64, 79, 80, 95, 96, 111, 113–126	9	0.07031
17–30, 33–46, 49–62, 65–78, 81–94, 97–110	10	0.07813

head presented by the application execution. We consider a value smaller than 5% for the overhead prediction error acceptable for those kind of predictions.

4.2.2.3 Adaptation to the Application's Characteristics

Model variables such as t_c and t_l depends on the amount of memory used by application processes. And processes on the same application may present different memory footprints. This occurs because processes compute different data or processes play different roles in the parallel application.

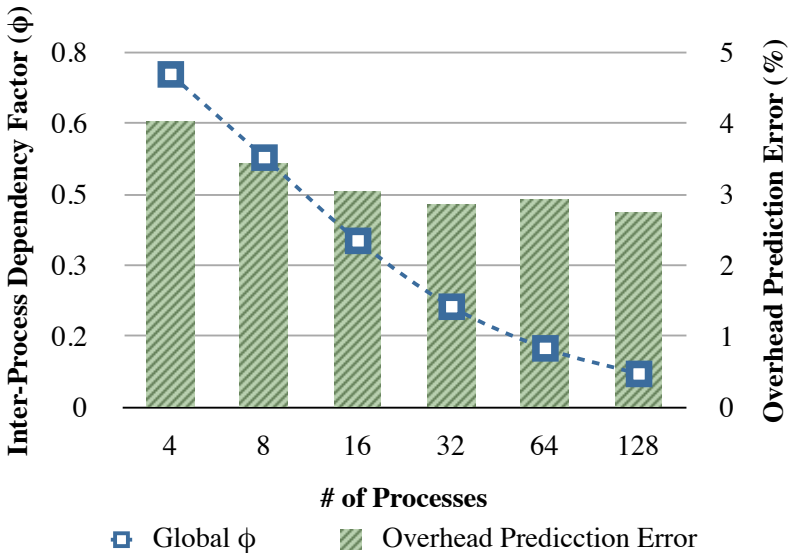


Figure 4.23: Overhead prediction error and the inter-process dependency factor for the LU applications according to the number of processes.

To depict the adaptation of the checkpoint interval to the process memory footprint the NAMD molecular dynamics application (70) have been used. NAMD is implemented over a Master/Worker paradigm where workers also communicate between themselves; the master process requires more memory in comparison to the workers.

The experiment has been executed using with a fault frequency (α) of 3600 seconds and the heartbeat frequency (t_d) was set to 1 second. Values for t_c, t_l, Δ_{lp}, and Δ_{lr} were measured during the execution. For this application we have manually calculated the inter-process dependency factor (ϕ) and its value is 1.

Dashed lines in figures 4.24 and 4.25 depict the checkpoint interval used throughout the application execution. Figure 4.24 refers to the master process, while figure 4.25 refers to a worker process. Figure 4.25 depicts only one worker processes, however others present a similar behaviour.

As the figures depict, processes use a small amount of memory in the

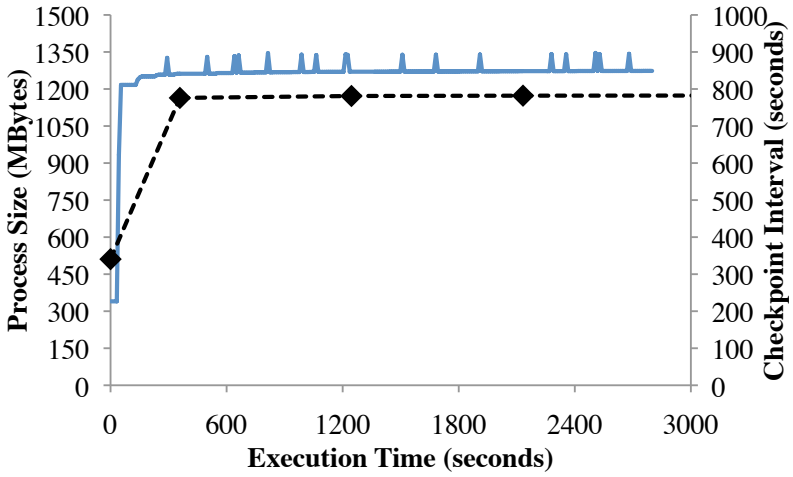


Figure 4.24: The continuous line shows the memory footprint of the NAMD master process running the "stmv" workload; values are shown on the left axes. The dashed line represents the checkpoint interval used; values are shown on the right axes. The rhombus points depict checkpoint instances.

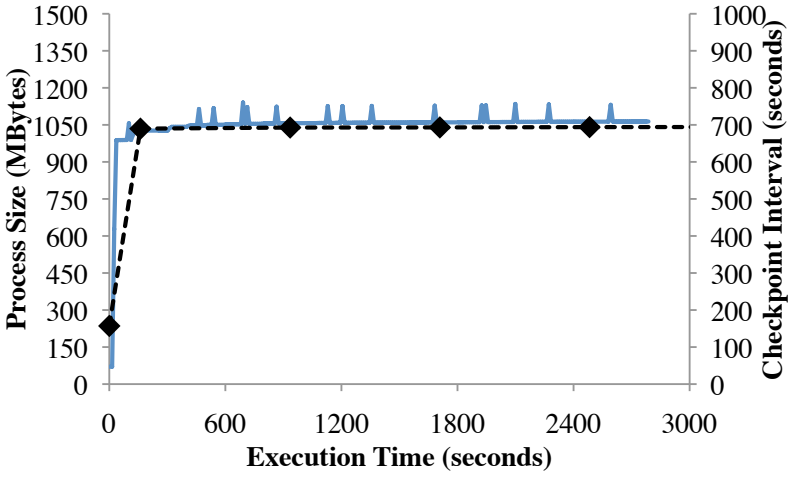


Figure 4.25: The continuous line shows the memory footprint of the a NAMD worker process running the "stmv" workload; values are shown on the left axes. The dashed line represents the checkpoint interval used; values are shown on the right axes. The rhombus points depict checkpoint instances.

startup phase. As a consequence of this the model calculates a short checkpoint interval initially. However, after the startup phase the application increases its memory footprint. After the second checkpoint the checkpoint interval changes to reflect the changes on the process memory footprint. Tables 4.14 and 4.15 summarise the checkpoint instances and sizes for the master and a worker process, respectively.

To depict the adaptation of the checkpoint to the inter-process dependency factor a dynamic matrix multiplication application built under a Master/Worker paradigm where workers only communicate with the master process have been used. This parallel application was executed using 8 nodes.

Considering the equation 3.17, the initial value for the ϕ variable is 0.34375. This value represents a global view of the relationship established between all processes on this parallel application. The fault frequency (α) has been defined as 3600 seconds and the heartbeat fre-

Table 4.14: First four values of the checkpoint size and the calculated next checkpoint interval for the NAMD master process running the "stmv" workload.

Execution Instant	Process Size	Checkpoint Interval
0.31 seconds	339.96 MB	340.56 seconds
358.45 seconds	1261.75 MB	775.93 seconds
1245.35 seconds	1270.05 MB	779.04 seconds
2132.25 seconds	1272.37 MB	779.41 seconds

Table 4.15: First four values of the checkpoint size and the calculated next checkpoint interval for a NAMD worker process running the "stmv" workload.

Execution Instant	Process Size	Checkpoint Interval
0.19 seconds	70.57 MB	157.35 seconds
160.95 seconds	1026.79 MB	689.77 seconds
934.97 seconds	1056.99 MB	693.29 seconds
1708.81 seconds	1060.23 MB	693.87 seconds

quency (t_d) has been set to 1 second. Values for t_c, t_l, Δ_{lp}, Δ_{lr}, and ϕ are measured during the execution. Equation 3.18 has been used to define in run-time the value of ϕ for each process.

Continuous lines in figures 4.26 and 4.27 depict the calculated values for the ϕ in run-time for the master and for a worker process, respectively. In addition, the dashed line on these figures depicts the values of the checkpoint interval during the application execution as well as the checkpoints instances.

As shown in figure 4.26, the initial value of 0.34375 was redefined to 1. This occurs because between the first and the second checkpoint the master process communicated with all 7 workers. As a consequence of this increase in the value of ϕ, the model has changed the checkpoint interval. Similarly, in figure 4.27 the decrease in the value of ϕ increases the time between checkpoints for a worker process.

Figure 4.27 depicts only one worker process, however, other worker processes present similar behaviour. The huge difference between the

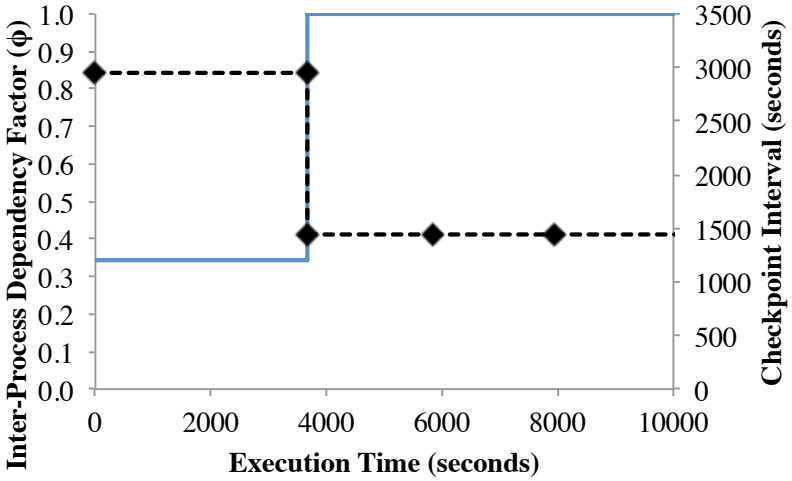


Figure 4.26: The continuous line shows the value of ϕ for the master process of the matrix multiplication; values are shown on the left axes. The dashed line represents the checkpoint interval used; values are shown on the right axes. The rhombus points depict checkpoint instances.

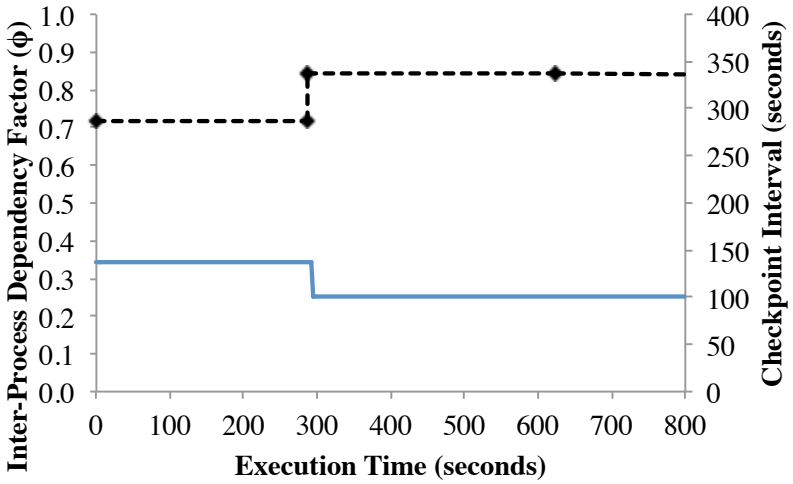


Figure 4.27: The continuous line shows the value of ϕ for a worker process of the matrix multiplication; values are shown on the left axes. The dashed line represents the checkpoint interval used; values are shown on the right axes. The rhombus points depict checkpoint instances.

checkpoint interval calculated for the master and for the worker process is caused by the difference in the memory footprint of these processes. Tables 4.16 and 4.17 summarise the checkpoint instances and sizes for the master and a worker process, respectively.

The next experiments depict the performance gain in using the methodology to define the checkpoint interval in run-time. This experiment compares the performance of our proposal with a static configuration in

Table 4.16: First five values of ϕ and the calculated next checkpoint interval for the master process of the matrix multiplication execution according to the execution instance.

Execution Instant	Process ϕ	Checkpoint Interval
0.84 seconds	0.34375	2,949.62 seconds
3,658.74 seconds	1.00000	1,436.16 seconds
5,823.01 seconds	1.00000	1,433.98 seconds
7,928.74 seconds	1.00000	1,439.21 seconds
10,054.94 seconds	1.00000	1,433.73 seconds

Table 4.17: First five values of ϕ and the calculated next checkpoint interval for a worker process of the matrix multiplication execution according to the execution instance.

Execution Instant	Process ϕ	Checkpoint Interval
0.31 seconds	0.34375	286.52 seconds
286.84 seconds	0.25000	336.68 seconds
623.52 seconds	0.25000	334.99 seconds
960.20 seconds	0.25000	337.09 seconds
1,296.88 seconds	0.25000	336.96 seconds

a faulty and fault-free scenario. The comparison was made using the aforementioned NAMD and dynamic matrix multiplication applications.

In these experiments only one fault was injected in each execution. The moment of the fault differs from one execution to other. The fault is distributed along the application execution according to the MT19937 PRNG algorithm. Each experiment has been executed at least 16 times and values are the average of all data that fall in a 95% confidence interval.

As shown in figure 4.28 the use of the fault tolerance provided by the RADIC/OMPI library introduces an overhead of about 25% in a fault-free execution and about 34% in a faulty scenario.

As shown in table 4.18 there is a modest reduction in the overhead while the checkpoint interval is calculated in run-time. This occurs because there is no significative change in the NAMD processes characteristics, except for the memory footprint in the start-up phase. However, as shown in figure 4.29 the matrix multiplication application presents

Table 4.18: Summary of the overhead experienced by NAMD and a matrix multiplication execution time using different fault tolerance configuration strategies on different environments.

App	**Statically Configured**		**Configured in Run-Time**	
	Fault-free	**Faulty**	**Fault-free**	**Faulty**
NAMD	25.4%	25.1%	34.5%	33.6%
MM	18.1%	24.6%	35.4%	31.1%

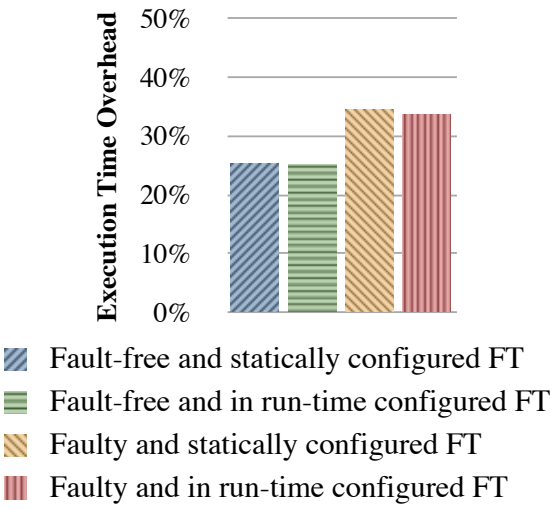


Figure 4.28: Comparison of the NAMD execution time using different fault tolerance configuration strategies on different environments.

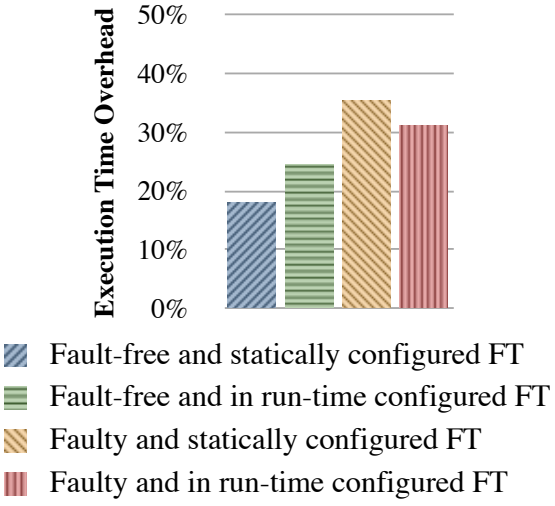


Figure 4.29: Comparison of the matrix multiplication execution time using different fault tolerance configuration strategies on different environments.

different results.

In the fault-free environment the execution using statically configured checkpoint interval presents a smaller overhead than the execution running with in run-time configuration. This is because the initial global

value of ϕ increases the checkpoint interval for the master process. This reduces the number of checkpoints performed. In this situation, the overhead introduced by a fault increases. This can be verified when we compare the total wall time clock in a faulty environment for the configurations made statically and in run-time.

4.2.3 Fault Tolerance Overhead

It is important to understand that the fault tolerance overhead is not only a question of the fault tolerance architecture configuration. It is a complex relation between the parallel application behaviour over the parallel computer. Moreover, the fault tolerance tasks may change the expected communication time as well as stop the computation phases for checkpointing. In addition, the injection of faults forces processes to rollback.

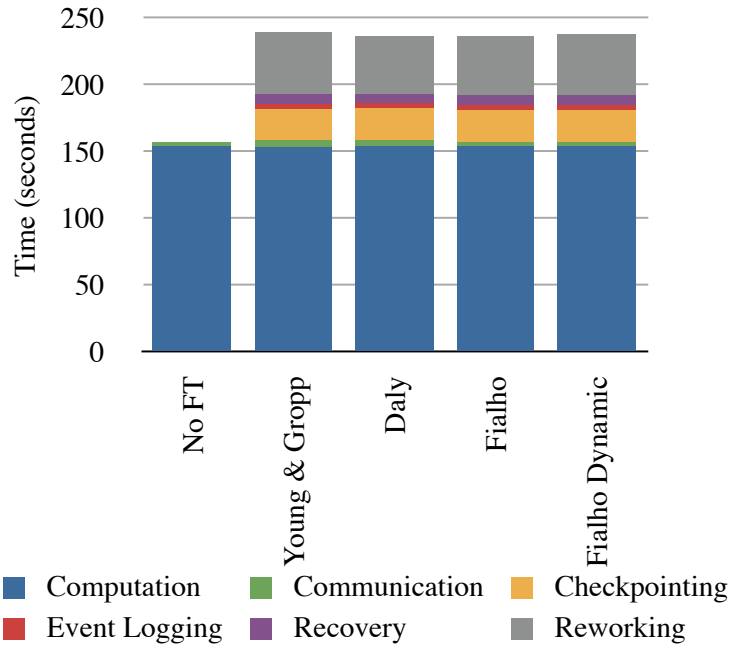
To understand how all these operations affects the overall parallel application execution time the following set of experiments were ran. The class B of the BT NAS benchmark have been chosen to depict this experiment. This application have been executed with 4, 9, 16, 25, and 36 processes. Executions ran in different number of nodes, depending on the number of processes. Some execution have been made using one process *per* node and other with four processes *per* node. For all these experiments the MTTI was set to 100 seconds.

Figure 4.30 shows the execution time breakdown of the NAS BT class B running with 4 processes. As a reference, the first column depicts the execution of this application without any sort of fault tolerance. Running one process *per* node, this application takes about to 157 seconds to run. Practically all the communication is overlapped with computation. The value of ϕ is 1 for this application under these circunstances.

As the value of the ϕ is 1, all models perform very similar. As well as the communication time, the event logging time can be partially over-

lapped with computation. The checkpointing and reworking time are the main responsible for the overhead introduced by the fault tolerant tasks.

Figure 4.31 depicts the same data of figure 4.30 while running the application with 9 processes. It is possible to notice the reduction on the computation time. As the number of peers each process communicate with increases. Thus, the application suffer a increase on the communi-

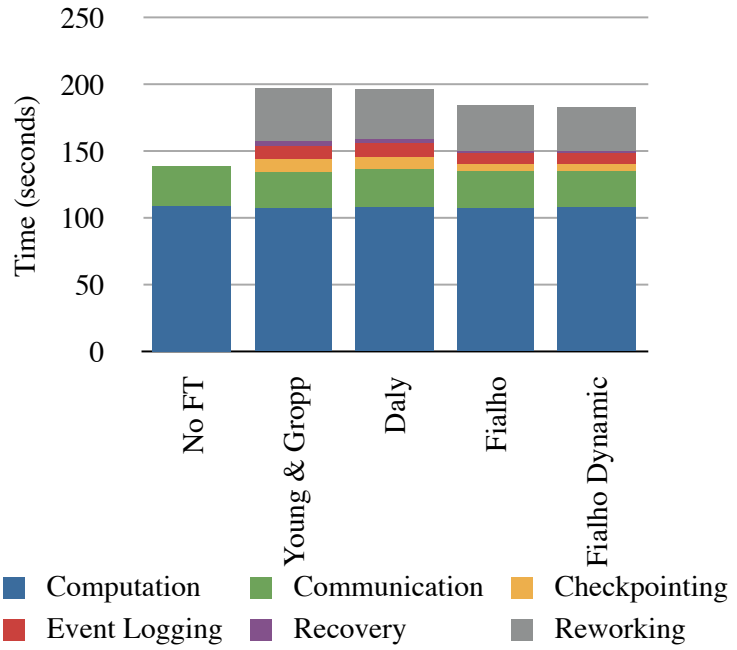


Task	No FT	Young & Gropp	Daly	Fialho	Fialho Dynamic
Computation	153.92	153.39	154.03	153.89	153.88
Communication	3.49	5.14	4.96	3.59	3.61
Checkpointing	—	23.50	23.61	23.85	23.94
Event Logging	—	3.51	3.45	3.69	3.49
Recovery	—	7.47	7.26	7.10	7.51
Reworking	—	46.28	42.96	44.21	45.09
Total	157.41	239.29	236.27	236.33	237.52

Figure 4.30: Execution time breakdown of the NAS BT class B according to the checkpoint interval model. Experiments ran with 4 processes, 1 process *per* node. Values are in seconds. $\phi = 1$, $\alpha = 100$.

cation time.

Under this circumstances the value of the global ϕ is close to 0.9, while there are three groups of processes with the value of ϕ varying from 1 to 0.77. This slight variation on the value of the ϕ reduces the checkpoint frequency as well as the interference of a faulty process on other processes. These modifications are noticeable on the checkpointing and reworking

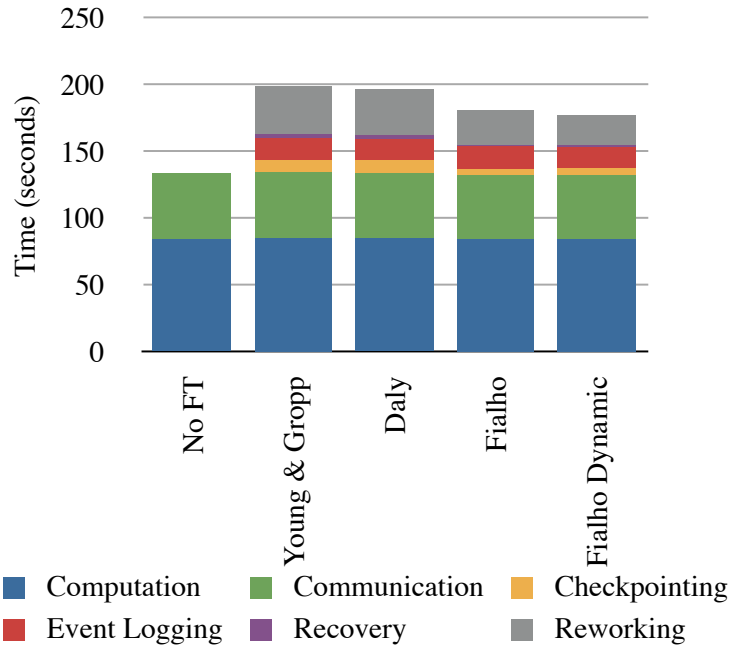


Task	No FT	Young & Gropp	Daly	Fialho	Fialho Dynamic
Computation	109.54	107.54	108.49	107.74	108.13
Communication	29.73	27.42	28.21	27.96	27.33
Checkpointing	—	9.31	9.42	5.01	5.10
Event Logging	—	10.18	10.08	8.43	8.70
Recovery	—	3.21	3.00	0.98	1.39
Reworking	—	40.26	37.38	34.48	32.46
Total	139.27	197.93	196.58	184.60	183.11

Figure 4.31: Execution time breakdown of the NAS BT class B according to the checkpoint interval model. Experiments ran with 9 processes, 1 process *per* node. Values are in seconds. $\phi \approx 0.90$, $\alpha = 100$.

time shown in table depicted in figure 4.31.

It is possible to notice a general reduction on the execution time for all models. Beside the lowering on the computation time, the main responsible for this is the diminution of the checkpointing time. It occurs because this scenario generates checkpoint files smaller than the previous one.

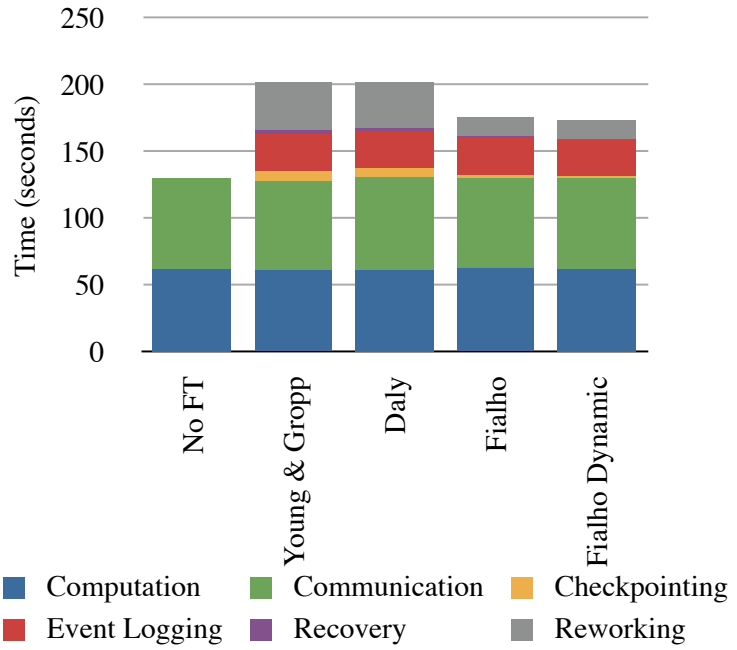


Task	No FT	Young & Gropp	Daly	Fialho	Fialho Dynamic
Computation	84.54	85.57	85.08	84.69	84.86
Communication	49.04	48.92	49.21	47.53	47.61
Checkpointing	—	9.12	9.23	4.91	4.97
Event Logging	—	16.21	16.11	16.51	16.17
Recovery	—	3.16	2.95	0.96	1.37
Reworking	—	36.10	33.94	26.53	22.55
Total	133.58	199.07	196.51	181.13	177.52

Figure 4.32: Execution time breakdown of the NAS BT class B according to the checkpoint interval model. Experiments ran with 16 processes, 1 process *per* node. Values are in seconds. $\phi \approx 0.59$, $\alpha = 100$.

The same tendency depicted from figure 4.30 to figure 4.31 can be perceived on figure 4.32 and figure 4.33. However, at this point, there is only a slight gain on the execution time. This occurs because the application starts to be driven by the communication time.

The event logging time increases because it depends on the application communication behaviour. The checkpointing and recovery times

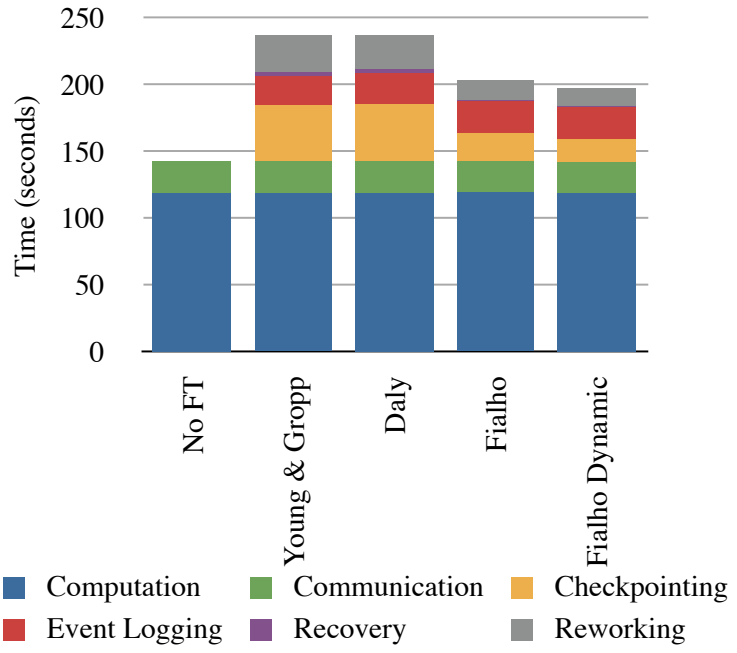


Task	No FT	Young & Gropp	Daly	Fialho	Fialho Dynamic
Computation	62.16	61.22	61.54	62.47	62.32
Communication	67.85	67.12	68.94	67.91	67.49
Checkpointing	—	7.30	7.41	2.19	1.87
Event Logging	—	27.48	27.38	27.78	27.44
Recovery	—	2.61	2.40	0.82	0.73
Reworking	—	36.10	33.94	14.59	13.53
Total	130.01	201.83	201.61	175.76	173.38

Figure 4.33: Execution time breakdown of the NAS BT class B according to the checkpoint interval model. Experiments ran with 25 processes, 1 process *per* node. Values are in seconds. $\phi \approx 0.45$, $\alpha = 100$.

diminish because the process memory footprint is smaller. And finally, as smaller is the value of the ϕ, less interference is generated by a faulty process on other processes. Another consequence of the diminution of the ϕ is a reduction on the checkpoint frequency that can be noticed by the brutal reduction on the checkpointing time.

Experiments depicted in figures 4.34 and 4.35 differ from the previ-

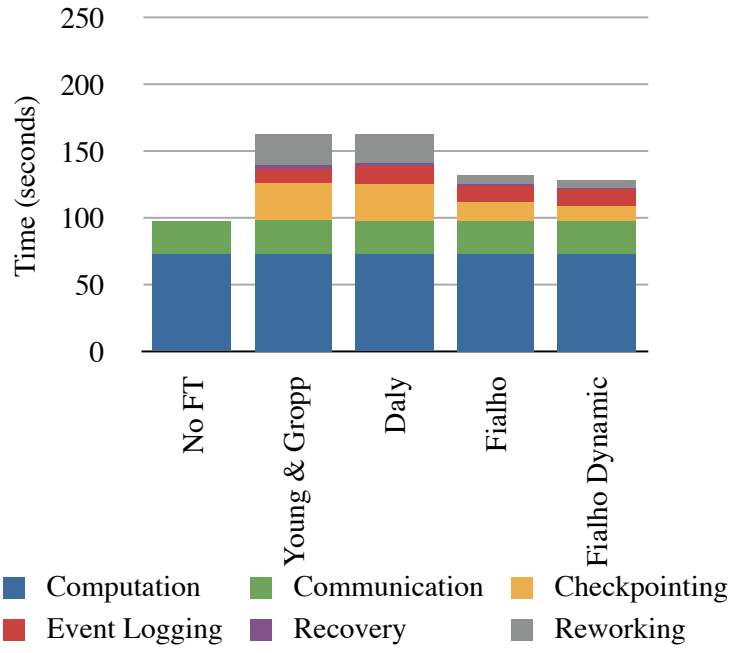


Task	No FT	Young & Gropp	Daly	Fialho	Fialho Dynamic
Computation	119.02	118.83	119.07	119.31	118.99
Communication	23.98	24.19	24.00	23.49	23.35
Checkpointing	—	41.82	41.93	21.26	17.17
Event Logging	—	21.39	23.73	23.40	23.29
Recovery	—	3.16	2.95	1.30	0.93
Reworking	—	27.77	25.78	14.59	13.53
Total	143.00	237.15	237.45	203.35	197.26

Figure 4.34: Execution time breakdown of the NAS BT class B according to the checkpoint interval model. Experiments ran with 16 processes, 4 process *per* node. Values are in seconds. $\phi \approx 0.59$, $\alpha = 100$.

ous because those two were executed with 4 processes *per* node. The main consequence of having more than one process *per* node concerns the competition for resources such as network, disk and access to the main memory. In this case the CPU is not a problem because nodes have not been overloaded with more processes than core exists on that.

Comparing figure 4.32 with figure 4.34 it is possible to notice that the



Task	No FT	Young & Gropp	Daly	Fialho	Fialho Dynamic
Computation	73.21	73.42	73.27	73.46	73.59
Communication	24.79	24.95	24.72	24.23	24.12
Checkpointing	—	27.90	28.01	14.30	11.60
Event Logging	—	10.66	13.00	12.67	12.56
Recovery	—	2.93	2.83	0.97	0.98
Reworking	—	23.14	21.48	6.63	5.86
Total	98.00	162.99	163.30	132.26	128.71

Figure 4.35: Execution time breakdown of the NAS BT class B according to the checkpoint interval model. Experiments ran with 36 processes, 4 process *per* node. Values are in seconds. $\phi \approx 0.30$, $\alpha = 100$.

computation time increases from 84.54 seconds to 119.02. This increment is explained by the concurrency on accessing the main memory, since the workload and the number of processes for both execution are the same.

Another significance difference concerns the apparently reduction on the communication time. Indeed there is a reduction, since some process communicate with other processes on the same node. The MPI library used to run these experiment this communication goes through a special communication channel implemented using shared memory. However the event logging overhead increases because those data should be stored on a remote node. In addition there are four processes using only one network to communicate.

The most evident consequence of sharing hardware resources can be perceived on the checkpointing time. There are two reasons for this increasing. The first one concerns the transference of the checkpoint to a remote node using a collapsed network. The second reason concerns the concurrency for the disk. Even so checkpoint are not coordinated it is possible that processes are performing the checkpoint at the same time.

Analysing figure 4.35, in comparison with figure 4.34 it is possible to notice the same changes while analysing previous figures. As usual, the value of the ϕ is smaller for 36 processes than for 16, which reduces the reworking time. Obviously the computation time suffer a evident diminution and the time of all fault tolerance tasks related to checkpoint files diminish as well.

The meaning of things lies not in the things themselves,
but in our attitude towards them.
— **Antoine de Saint-Exupéry**

5

Conclusion

This thesis has presented a novel model to face the emergent fault tolerance paradigm. This paradigm tries to provide a fault tolerance infrastructure suitable for scalable parallel applications. To protect these applications the fault tolerance architecture relies on uncoordinated checkpointing combined with event logging techniques.

There was a lack of knowledge concerning the use of protocols that combine uncoordinated checkpointing with event logging. This thesis has presented a model to estimate the wall time clock of a parallel application protected by uncoordinated checkpointing. As well as a convenient model to calculate the frequency in which those checkpoints should be taken.

This thesis has demonstrated that previous models are not useful to define an accurate checkpoint interval for parallel applications protected by uncoordinated checkpointing protocols. Furthermore, as the number of processes in the parallel application increases the accuracy of those models tends to decrease.

On the development of this work a model for coordinated checkpointing have been presented. After that, a model for parallel applications has been introduced. A key point is the modelling of the relationship between parallel application processes to define the checkpoint interval for uncoordinated checkpointing protocols. According to this study, the

relationship between processes is defined by the communication events exchanged between the application processes. This relationship was called inter-process dependency factor.

Some applications characteristics are important to model the relationship existent between parallel application processes like: peers each process communicates with, frequency of this communication, the domino effect caused by a fault, and the time needed to recover a faulty process.

A first order approximation to the definition of the inter-process dependency factor as well as a simple methodology to define the variables of this factor was presented and are the following:

$$\phi = \frac{P(n)}{N}$$

were ϕ should be defined individually for each application process as well as the checkpoint interval. In this first approximation the $P(n)$ function considers only the peers each process communicates with.

The checkpoint interval model that minimises the overhead introduced by the fault tolerance tasks can be calculated using the following equation:

$$\sigma_{opt} = \frac{\sqrt{\phi t_c(2\alpha - t_c - 2\Delta_{lr})}}{\phi} - t_c$$

and the cost function useful to estimate the application runtime is depicted underneath:

$$T_{est} = T_p\left[1 + \frac{\phi\sigma^2 + \sigma(3\phi t_c + 2\phi\Delta_{lr} - t_c + 2\Delta_{lp})}{\alpha(2\sigma + 2t_c)} + \frac{2t_c(\phi t_c + \phi\Delta_{lr} + \alpha - t_c - \Delta_{lr} + \Delta_{lp})}{\alpha(2\sigma + 2t_c)}\right]$$

The coordinated checkpointing model presents a difference of less than 1.2% from other models on average and the execution time prediction error is smaller than 3% in comparison with a real application execution.

With regard to the parallel model, it presents an overhead prediction error smaller than 5% for the applications tested. Furthermore, we have demonstrated that our models perform better when the number of processes increases and there is less dependency between processes.

Moreover, this thesis have presented a methodology to dynamically define the input variables used by models based on measurements performed during the application execution. A methodology to monitor the processes that compose the parallel application has been proposed. To monitoring mechanism permits the definition of the variables value used in the checkpoint interval model.

This instrumentation allows the definition of the checkpoint interval in run-time with a high degree of precision, process by process. The use of this methodology reduces in about 3% the overhead introduced in the execution time for applications running in faulty environments.

5.1 Summary of Contributions

The contributions of this work have been accepted for publications on different conferences.

The first one (42) describes the MPI library that incorporates the RADIC fault tolerance architecture. This work has been published in the proceedings of the 16th European PVM/MPI user's group meeting in Helsinki, Finland.

The modelling of the relationship between parallel application processes to define the checkpoint interval for uncoordinated checkpointing protocols has been accepted for publication on the 31th International Conference on Distributed Computing Systems (41), to be held in Minneapolis, United States.

The definition of the checkpoint interval model for uncoordinated checkpointing protocols (39) as well as the methodology to define the

models variable values in run-time (40) have been published in the International Conference on Parallel and Distributed Processing Techniques and Applications, to be held in Las Vegas, United States.

5.2 Future Work

The overhead added to the application execution by the monitoring mechanism tends to be unappreciable. However, it is necessary to quantify this overhead.

The use of uncoordinated checkpointing is the only solution that allows the use of different checkpoint intervals for each application process. However, the use of uncoordinated checkpointing assisted by message logging may not be the solution that presents the lowest overhead. There is the need to analyse if a sender-based message logging or a coordinated checkpointing solution present better results.

Although good results can be achieved by the probability function presented in this paper, we consider this function very simple to define the relationship existent between the processes of a parallel application. More elaborate $P(n)$ functions can be designed. A good choice is to define $P(n)$ analysing the probability that a fault process blocks other processes during the recovery time (T_r).

Another important issue is to prove that the parallel model is suitable for libraries that implement uncoordinated check- pointing combined with sender-based message logging.

References

[1] A Agbaria and R Friedman. **Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations**. *Cluster Computing*, **6**(3):227—236, 2003. Available from: http://dx.doi.org/10.1023/A:1023540604208. 19

[2] A Agbaria and R Friedman. **Model-based performance evaluation of distributed checkpointing protocols**. *International Journal of Performance Evaluation*, **65**(5):345—365, 2008. Available from: http://dx.doi.org/10.1016/j.peva.2007.09.001. 23

[3] L Alvisi, B Hoppe, and Keith Marzullo. **Nonblocking and Orphan-Free Message Logging Protocols**. *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing*, pages 145—154, 1993. Available from: http://dx.doi.org/10.1109/FTCS.1993.627318. 22

[4] L Alvisi and K Marzullo. **Trade-Offs in Implementing Optimal Message Logging Protocols**. *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 58—67, 1996. Available from: http://dx.doi.org/10.1145/248052.248061. 14

[5] L Alvisi and K Marzullo. **Message Logging: Pessimistic, Optimistic, Causal, and Optimal**. *IEEE Transactions on Software Engineering*, **24**(2):149—159, 1998. Available from: http://dx.doi.org/10.1109/32.666828. 19

[6] M Aminian, M Akbari, and B Javadi. **Coordinated Checkpoint from Message Payload in Pessimistic Sender-Based Message Logging**. *Proceedings of the 20th International Parallel and Distributed Processing Symposium*, 2006. Available from: http://dx.doi.org/10.1109/IPDPS.2006.1639619. 23

[7] J Ansel, K Arya, and G Cooperman. **DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop**. *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, pages 1—12, 2009. Available from: http://dx.doi.org/10.1109/IPDPS.2009.5161063. 21

[8] S Arunagiri, J Daly, P Teller, S Seelam, R A Oldfield, M R Varela, and R Riesen. **Opportunistic Checkpoint Intervals to Improve System Performance**. *Technical Report: UTEP-CS-08-24*, 2008. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.147.3874. 22

[9] R Aulwes, D Daniel, N Desai, R Graham, L Dean Risinger, Mark A Taylor, Timothy S Woodall, and Mitchel W Sukalski. **Architecture of LA-MPI, a Network-Fault-Tolerant MPI**. *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004. Available from: http://dx.doi.org/10.1109/IPDPS.2004.1302920. 19

[10] R Baldoni, J Hélary, A Mostefaoui, and M Raynal. **Consistent Checkpointing in Message Passing Dstributed Systems**. *INRIA Technical Report*, 1995. Available from: http://hal.inria.fr/inria-00074117/en/. 13

[11] R Baldoni, J Hélary, A Mostefaoui, and M Raynal. **On Modeling Consistent Checkpoints and the Domino Effect in Distributed Systems**. *INRIA Technical Repport*, 1995. Available from: http://hal.inria.fr/inria-00074112/en/. 13

[12] 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, 2004. Available from: http://dx.doi.org/10.1023/B:CLUS.0000039491.64560.8a. 19

[13] R Batchu, J Neelamegam, Z Cui, and M Beddhu. **MPI/FT: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing**. *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 26—33, 2001. Available from: http://dx.doi.org/10.1109/CCGRID.2001.923171. 19

[14] 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**. *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1—18, Jan 2002. Available from: http://dx.doi.org/10.1109/SC.2002.10048. 6, 19

[15] A Bouteiller, G Bosilca, and J Dongarra. **Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging**. *Proceedings of the 2007 Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 297—306, 2007. Available from: http://dx.doi.org/10.1007/978-3-540-75416-9_41. 14

[16] A Bouteiller, G Bosilca, and J Dongarra. **Redesigning the Message Logging Model for High Performance**. *Concurrency and Computation: Practice and Experience*, **22**(16):2196—2211, 2010. Available from: http://dx.doi.org/10.1002/cpe.1589. 22

[17] A Bouteiller, F Cappello, T Herault, G Krawezik, P Lemarinier, and F Magniette. **MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging**. *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 25, Jan 2003. Available from: http://dx.doi.org/10.1109/SC.2003.10027. 6, 19

[18] A Bouteiller, B Collin, T Herault, P Lemarinier, and F Cappello. **Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI**. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pages 97—106, 2005. Available from: http://dx.doi.org/10.1109/IPDPS.2005.249. 14

[19] A Bouteiller, T Herault, G Krawezik, P Lemarinier, and F Cappello. **MPICH-V Project: A Multiprotocol Automatic Fault-Tolerant MPI**. *International Journal of High Performance Computing Applications*, **20**(3):319—333, 2006. Available from: http://dx.doi.org/10.1177/1094342006067469. 19, 23, 45

[20] A Bouteiller, P Lemarinier, K Krawezik, and F Capello. **Coordinated checkpoint versus message log for fault tolerant MPI**. *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, pages 242—250, 2003. Available from: http://dx.doi.org/10.1109/CLUSTR.2003.1253321. 23

[21] 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**. *Proceedings of the 2009 International Conference on Cluster Computing*, pages 1—9, 2009. Available from: http://dx.doi.org/10.1109/CLUSTR.2009.5289157. 22

[22] F Cappello. **Fault Tolerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities**. *International Journal of High Performance Computing Applications*, pages 212—226, 2009. Available from: http://dx.doi.org/10.1177/1094342009106189. 6, 21, 23

[23] S Chakravorty and L Kalé. **A Fault Tolerant Protocol for Massively Parallel Systems**. *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pages 212—219, 2004. Available from: http://dx.doi.org/10.1109/IPDPS.2004.1303244. 23

[24] K Chandy and L Lamport. **Distributed Snapshots: Determining Global States of Distributed Systems**. *ACM Transactions on Computer Systems (TOCS)*, **3**(1):63—75, 1985. Available from: http://dx.doi.org/10.1145/214451.214456. 12

[25] K Chandy and C Ramamoorthy. **Rollback and Recovery Strategies for Computer Programs**. *IEEE Transactions on Computers*, **21**(6):546—556, 1972. Available from: http://dx.doi.org/10.1109/TC.1972.5009007. 1

[26] Y Chen, J Plank, and K Li. **CLIP: A Checkpointing Tool for Message-Passing Parallel Programs**. *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, 1997. Available from: http://dx.doi.org/10.1145/509593.509626. 21

[27] M Chung and M Krishnamoorthy. **Algorithms of Placing Recovery Points**. *Information Processing Letters*, **28**(4):177—181, 1988. Available from: http://dx.doi.org/10.1016/0020-0190(88)90205-0. 20

[28] C Coti, T Herault, P Lemarinier, and L Pilard. **Blocking vs. Non-Blocking Coordinated Checkpointing for Large-Scale Fault Tolerant MPI**. *Proceedings of the ACM/IEEE 2006 Conference on Supercomputing*, 2006. Available from: http://dx.doi.org/10.1145/1188455.1188587. 22

[29] J Daly. **A higher order estimate of the optimum checkpoint interval for restart dumps**. *Future Generation Computer Systems*, **22**(3):303—312, 2006. Available from: http://dx.doi.org/10.1016/j.future.2004.11.016. 2, 20, 28, 30

[30] O Damani, Y M Wang, and V Garg. **Distributed recovery with K-optimistic logging**. *Journal of Parallel and Distributed Computing*, **63**(12):1193—1218, 2003. Available from: http://dx.doi.org/10.1016/j.jpdc.2003.07.003. 23

[31] A Duarte. **RADIC: A Powerful Fault-Tolerant Architecture**. *PhD Thesis*, 2007. Available from: http://www.recolecta.net/buscador/single_page.jsp?id=oai:UAB.es:TDX-1126107-101303. 23

[32] A Duarte, D Rexachs, and E Luque. **A Distributed Scheme for Fault-Tolerance in Large Clusters of Workstations**. *Proceedings of the 2005 International Conference Parallel Computing*, **33**:473—480, 2006. Available from: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.11&rep=rep1&type=pdf. 23

[33] A Duarte, D Rexachs, and E Luque. **Increasing the cluster availability using RADIC**. *Proceedings of the 2006 IEEE International Conference on*

Cluster Computing, 2006. Available from: http://dx.doi.org/10.1109/CLUSTR.2006.311872. 53, 54

[34] E Elnozahy, L Alvisi, Y Wang, and D Johnson. **A Survey of Rollback-Recovery Protocols in Message-Passing Systems**. *ACM Computing Surveys*, **34**(3):375—408, 2002. Available from: http://dx.doi.org/10.1145/568522.568525. 3, 10, 34, 43

[35] E Elnozahy and J Plank. **Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery**. *IEEE Transactions on Dependable and Secure Computing*, **1**(2):97—108, 2004. Available from: http://dx.doi.org/10.1109/TDSC.2004.15. 23

[36] G Fagg, T Angskun, G Bosilca, and J Pjesivac-Grbovic. **Scalable Fault Tolerant MPI: Extending the Recovery Algorithm**. *Proceedings of the 12th European PVM/MPI Users' Group Meeting*, pages 67—75, 2005. Available from: http://dx.doi.org/10.1007/11557265_13. 23

[37] G Fagg, A Bukovsky, and J Dongarra. **HARNESS and fault tolerant MPI**. *Parallel Computing*, **27**(11):1479—1495, 2001. Available from: http://dx.doi.org/10.1016/S0167-8191(01)00100-4. 19

[38] G Fagg and J Dongarra. **FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World**. *Proceedings of the 7th European PVM/MPI Users' Group Meeting*, pages 346—353, 2000. Available from: http://dx.doi.org/10.1007/3-540-45255-9_47. 19

[39] L Fialho, D Rexachs, and E Luque. **Defining the Checkpoint Interval for Uncoordinated Checkpointing Protocols**. **To appear in** *the Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, page to appear, 2011. 95

[40] L Fialho, D Rexachs, and E Luque. **On the Calculation of the Checkpoint Interval in Run-Time for Parallel Applications**. **To appear in** *the Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, page to appear, 2011. 96

[41] L Fialho, D Rexachs, and E Luque. **What Is Missing in Current Checkpoint Interval Models? To appear in** *the Proceedings of the 31th International Conference on Distributed Computing Systems*, page to apper, 2011. 20, 95

[42] L Fialho, G Santos, A Duarte, D Rexachs, and E Luque. **Challenges and Issues of the Integration of RADIC into Open MPI**. *Proceedings of the 16th European PVM/MPI Users' Group Meeting*, pages 73—83, Jan 2009.

Available from: http://dx.doi.org/10.1007/978-3-642-03770-2_14. 6, 53, 54, 95

[43] R Gioiosa, J Sancho, S Jiang, F Petrini, and Kei Davis. **Transparent, Incremental Checkpointing at Kernel Level: a Foundation for Fault Tolerance for Parallel Computers**. *Proceedings of the ACM/IEEE 2005 Conference on Supercomputing*, 2005. Available from: http://dx.doi.org/10.1109/SC.2005.76. 21

[44] W Gropp and E Lusk. **Fault Tolerance in Message Passing Interface Programs**. *International Journal of High Performance Computing Applications*, **18**(3):363—372, 2004. Available from: http://dx.doi.org/10.1177/1094342004046045. 2, 20, 30, 32

[45] R Gupta, H Naik, and P Beckman. **Understanding Checkpointing Overheads on Massive-Scale Systems: Analysis on the IBM Blue Gene/P System**. *International Journal of High Performance Computing Applications*, 2010. Available from: http://dx.doi.org/10.1177/1094342010369118. 22

[46] T 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. Available from: http://dx.doi.org/10.1016/j.jpdc.2009.03.007. 1

[47] P Hargrove and J Duell. **Berkeley lab checkpoint/restart (BLCR) for Linux clusters**. *Journal of Physics: Conference Series*, **46**:494—499, 2006. Available from: http://dx.doi.org/10.1088/1742-6596/46/1/067. 21

[48] J Hursey, J Squyres, T Mattox, and A Lumsdaine. **The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI**. *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium*, Jan 2007. Available from: http://dx.doi.org/10.1109/IPDPS.2007.370605. 6, 19

[49] W Jones, JT Daly, and N DeBardeleben. **Impact of Sub-optimal Checkpoint Intervals on Application Efficiency in Computational Clusters**. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 276—279, 2010. Available from: http://dx.doi.org/10.1145/1851476.1851509. 20

[50] S Kalaiselvi and V Rajaraman. **A survey of checkpointing algorithms for parallel and distributed computers**. *Sadhana*, **25**(5):489—510, 2000. Available from: http://dx.doi.org/10.1007/BF02703630. 10

[51] R Koo and S Toueg. **Checkpointing and Rollback-Recovery for Distributed Systems**. *IEEE Transactions of Software Engineering*, **13**(1):23–31, 1987. Available from: http://dx.doi.org/10.1109/TSE.1987.232562. 12

[52] A Kshemkalyani, M Raynal, and M Singhal. **An introduction to snapshot algorithms in distributed computing**. *Distributed Systems*, **2**:224—233, 1995. Available from: http://dx.doi.org/10.1088/0967-1846/2/4/005. 12

[53] L Lamport. **Time, Clocks, and the Ordering of Events in a Distributed System**. *Communications of the ACM*, **21**(7):558—565, 1978. Available from: http://dx.doi.org/10.1145/359545.359563. 14

[54] P Lemarinier, A Bouteiller, T Herault, G Krawezik, and F Cappello. **Improved Message logging versus Improved coordinated checkpointing for fault tolerant MPI**. *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, pages 115—124, 2004. Available from: http://dx.doi.org/10.1109/CLUSTR.2004.1392609. 23

[55] C Li, E Stewart, and W Fuchs. **Compiler-Assisted Full Checkpointing**. *Software: Practice and Experience*, **24**(10):871—886, 1994. Available from: http://dx.doi.org/10.1002/spe.4380241002. 22

[56] W Li and J Tsay. **Checkpointing Message-Passing Interface (MPI) Parallel Programs**. *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 147—152, 1997. Available from: http://dx.doi.org/10.1109/PRFTS.1997.640140. 10

[57] Y Ling, J Mi, and X Lin. **A Variational Calculus Approach to Optimal Checkpoint Placement**. *IEEE Transactions on Computers*, **50**(7):699—708, 2001. Available from: http://dx.doi.org/10.1109/12.936236. 20

[58] J Long, W Fuchs, and J Abraham. **Compiler-Assisted Static Checkpoint Insertion**. *Proceedings of the 1992 International Symposium on Fault-Tolerant Computing*, pages 58—65, 1992. Available from: http://dx.doi.org/100.2/ADA256039. 22

[59] S Louca, N Neophytou, A Lachanas, and P Evripidou. **MPI-FT: Portable Fault Tolerance Scheme for MPI**. *Parallel Processing Letters*, **10**(4):371—382, 2000. Available from: http://dx.doi.org/10.1142/S0129626400000342. 19

[60] M Matsumoto and T Nishimura. **Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator**. *ACM Transactions on Modeling and Computer Simulation*, **8**(1):3—30, 1998. Available from: http://dx.doi.org/10.1145/272991.272995. 50

[61] H Nam, J Kim, S Hong, and S Lee. **Probabilistic Checkpointing**. *Proceedings of the 27th Annual International Symposium on Fault-Tolerant Computing*, pages 48—57, 1997. Available from: http://dx.doi.org/10.1109/FTCS.1997.614077. 21

[62] V Nicola and JV Spanje. **Comparative Analysis of Different Models of Checkpointing and Recovery**. *IEEE Transactions on Software Engineering*, **16**(8):807—821, 1990. Available from: http://dx.doi.org/10.1109/32.57620. 10

[63] R Oldfield. **Investigating Lightweight Storage and Overlay Networks for Fault Tolerance**. *Procceding of the 2006 High Availability and Performance Computing Workshop*, 2006. Available from: http://xcr.cenit.latech.edu/hapcw2006/program/papers/lwfs-overlay.pdf. 21

[64] R A Oldfield, S Arunagiri, P J Teller, S Seelam, M R Varela, R Riesen, and P C Roth. **Modeling the Impact of Checkpoints on Next-Generation Systems**. *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 30—46, 2007. Available from: http://dx.doi.org/10.1109/MSST.2007.4367962. 23

[65] A Oliner, L Rudolph, and R Sahoo. **Cooperative Checkpointing: A Robust Approach to Large-Scale Systems Reliability**. *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 14—23, 2006. Available from: http://dx.doi.org/10.1145/1183401.1183406. 22

[66] A Oliner and R Sahoo. **Evaluating Cooperative Checkpointing for Supercomputing Systems**. *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, 2006. Available from: http://dx.doi.org/10.1109/IPDPS.2006.1639693. 22

[67] A Oliner, R Sahoo, J Moreira, and M Gupta. **Performance Implications of Periodic Checkpointing on Large-Scale Cluster Systems**. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, pages 299—306, 2005. Available from: http://dx.doi.org/10.1109/IPDPS.2005.337. 22

[68] T Ozaki, T Dohi, H Okamura, and N Kaio. **Min-Max Checkpoint Placement Under Incomplete Failure Information**. *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pages 721—-730, 2004. Available from: http://dx.doi.org/10.1109/DSN.2004.1311943. 20

[69] H S Paul, A Gupta, and A Sharma. **Finding a suitable checkpoint and recovery protocol for a distributed application**. *Journal of Parallel and*

Distributed Computer, **66**(5):732—749, 2006. Available from: http://dx.doi.org/10.1016/j.jpdc.2005.12.008. 23

[70] J C Philips, R Braun, W Wang, J Gumbart, E Tajkhorshid, E Villa, C Chipot, R D Skeel, L Kalé, and K Schulten. **Scalable Molecular Dynamics with NAMD**. *Journal of Computational Chemistry*, **26**(16):1781—1802, 2005. Available from: http://dx.doi.org/10.1002/jcc.20289. 78

[71] S Rao, L Alvisi, and H Vin. **Egida: An Extensible Toolkit For Low-overhead Fault-Tolerance**. *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 48—55, 1999. Available from: http://dx.doi.org/10.1109/FTCS.1999.781033. 19

[72] S Rao, L Alvisi, and H Vin. **The Cost of Recovery in Message Logging Protocols**. *IEEE Transactions on Knowledge and Data Engineering*, **12**(2):160—173, 2000. Available from: http://dx.doi.org/10.1109/69.842260. 14, 43, 69

[73] T Ropars and C Morin. **O2P: An Extremely Optimistic Message Logging Protocol**. *INRIA Research Report 6357*, 2007. Available from: http://hal.archives-ouvertes.fr/docs/00/18/78/79/PDF/RR-6357.pdf. 22

[74] J Sancho, F Petrini, K Davis, R Gioiosa, and S Jiang. **Current Practice and a Direction Forward in Checkpoint/Restart Implementations for Fault Tolerance**. *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005. Available from: http://dx.doi.org/10.1109/IPDPS.2005.157. 21

[75] S Sankaran, J Squyres, B Barrett, and V Sahay. **The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing**. *International Journal of High Performance Computing Applications*, **19**(4):479—493, 2005. Available from: http://dx.doi.org/10.1177/1094342005056139. 19

[76] G Santos, L Fialho, D Rexachs, and E Luque. **Increasing the Availability Provided by RADIC with Low Overhead**. *Proceedings of the 2009 IEEE International Conference on Cluster Computing*, pages 1—8, 2008. Available from: http://dx.doi.org/10.1109/CLUSTR.2009.5289163. 22

[77] W Saphir, R Wijngaart, A Woo, and M Yarrow. **New Implementations and Results for the NAS Parallel Benchmarks 2**. *Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, Jan 1997. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.43.3199. 67, 73, 75

[78] B Schroeder and G Gibson. **A large-scale study of failures in high-performance computing systems**. *Proceedings of the International Conference*

on Dependable Systems and Networks, pages 249—258, 2006. Available from: http://dx.doi.org/10.1109/DSN.2006.5. 1

[79] B Schroeder and G Gibson. **Understanding Failures in Petascale Computers**. *Journal of Physics: Conference Series*, **78**, 2007. Available from: http://dx.doi.org/10.1088/1742-6596/78/1/012022. 1

[80] T Sherwood, E Perelman, G Hamerly, and Brad Calder. **Automatically Characterizing Large Scale Program Behavior**. *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 45—57, 2002. Available from: http://dx.doi.org/10.1145/605397.605403. 48

[81] K Venkatesh, T Radhakrishnan, and H Li. **Optimal Checkpointing and Local Recording for Domino-Free Rollback Recovery**. *Information Processing Letters*, **25**(5):295—303, 1987. Available from: http://dx.doi.org/10.1016/0020-0190(87)90203-1. 13

[82] L Wang, K Pattabiraman, Z Kalbarczyk, R K Iyer, L Votta, C Vick, and A Wood. **Modeling Coordinated Checkpointing for Large-Scale Supercomputers**. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 812—821, 2005. Available from: http://dx.doi.org/10.1109/DSN.2005.67. 23

[83] A Wong, D Rexachs, and E Luque. **Parallel Application Signature**. *Proceedings of the 2009 IEEE International Conference on Cluster Computing*, 2009. Available from: http://dx.doi.org/10.1109/CLUSTR.2009.5289132. 40

[84] K Wong and M Franklin. **Distributed Computing Systems and Checkpointing**. *Proceedings of the 2nd International Symposium on High Performance Distributed Computing*, pages 224—233, 1993. Available from: http://dx.doi.org/10.1109/HPDC.1993.263838. 20

[85] J Yang, K F Li, W Li, and D Zhang. **Trading off logging overhead and coordinating overhead to achieve efficient rollback recovery**. *Concurrency Computat.: Pract. Exper.*, **21**(6):819—853, 2009. Available from: http://dx.doi.org/10.1002/cpe.1364. 23

[86] J Young. **A First Order Approximation to the Optimum Checkpoint Interval**. *Communications of the ACM*, **17**(9):530—531, 1974. Available from: http://dx.doi.org/10.1145/361147.361115. 2, 19, 30, 43

[87] G Zheng, L Shi, and L Kali. **FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI**.

Proceedings of the 2004 IEEE International Conference on Cluster Computing, pages 93—103, 2004. Available from: `http://dx.doi.org/10.1109/CLUSTR.2004.1392606`. 21

[88] H Zhong and J Nieh. **CRAK: Linux Checkpoint/Restart As a Kernel Module**. *Technical Report CUCS-014-01*, 2001. Available from: `http://www.cs.columbia.edu/techreports/cucs-014-01.pdf`. 21