Checkpointing of Parallel Applications in a Grid Environment

Kreeteeraj Sajadah

School of Electronics and Computer Science

Checkpointing of Parallel Applications in a Grid Environment

A dissertation submitted to the University of Westminster for the degree of Master of Philosophy

Kreeteeraj SAJADAH

Supervised by Dr. G. Z. Terstyanszky, Prof. S. C. Winter & Prof. P. Kacsuk

December 2011

Centre for Parallel Computing,
School of Electronics and Computer Science,
University of Westminster,
London, United Kingdom

To my dear and beautiful wife Jayshree, my adoring son Hevanshraj and my loving parents, Asween Coomar and Vedprabha Sajadah

Acknowledgements

First and foremost I offer my sincerest gratitude to Dr. Gabor Terstyanszky who has given me the possibility of starting my research in this institution offering his supervision since the beginning of my study. He provided me with many helpful suggestions, important advice and constant encouragement during the course of this work.

I would also like to express profound gratitude to my advisors, Professor Stephen Winter and Professor Peter Kacsuk for their invaluable support, supervision and useful suggestions. They were always there to meet and talk about my ideas, to proofread and mark up my work. Special thanks go to Dr. Gabor Terstyanszky for organizing my many research meetings, for chasing me up during my studies, and for his prompt corrections and suggestions during the writing up of my thesis.

I wish to acknowledge my friends and colleagues from the Centre for Parallel Computing for supporting me through my research work.

Many thanks are due to all those who read this document and spent hours helping me amassing the information used here.

And finally, I wish to thank all my family and beloved friends for all these singular years.

Abstract

The Grid environment is generic, heterogeneous, and dynamic with lots of unreliable resources making it very exposed to failures. The environment is unreliable because it is geographically dispersed involving multiple autonomous administrative domains and it is composed of a large number of components. Examples of failures in the Grid environment can be: application crash, Grid node crash, network failures, and Grid system component failures. These types of failures can affect the execution of parallel/distributed application in the Grid environment and so, protections against these faults are crucial. Therefore, it is essential to develop efficient fault tolerant mechanisms to allow users to successfully execute Grid applications. One of the research challenges in Grid computing is to be able to develop a fault tolerant solution that will ensure Grid applications are executed reliably with minimum overhead incurred.

While checkpointing is the most common method to achieve fault tolerance, there is still a lot of work to be done to improve the efficiency of the mechanism. This thesis provides an in-depth description of a novel solution for checkpointing parallel applications executed on a Grid. The checkpointing mechanism implemented allows to checkpoint an application at regions where there is no interprocess communication involved and therefore reducing the checkpointing overhead and checkpoint size.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

1 Introduction

1.1 Introduction

Grid computing can be considered to be a generalised version of the metacomputer combining the supercomputing technology with networking and web technology. A metacomputer is a collection of any kind of computers that are heterogeneous, geographically distributed, connected by a wide-area network and that appears and acts as a single computer to an individual. Grid computing has emerged as an important new field focusing on large-scale sharing and high-performance which distinguishes it from conventional distributed computing systems **[20].** The Grid environment utilise available resources. This can be in the form of dedicated resources or under used machines over the Internet.

The main difference between Grid computing and conventional high performance computing systems such as cluster computing is that Grids tend to be more loosely coupled, heterogeneous, and geographically dispersed **[1].**

At the very base, Grid computing is a computer network where each computer's resources are shared with every other computer in the system. Authorized users are able to access resources like processing power, memory and data storage to execute specific tasks. The complexity of a Grid computing system varies. We can either have a system that is as simple as a collection of similar computers running on the same operating system or as complex as inter-networked systems comprised of different computer platforms **[1].**

In Grid computing, a Virtual Organization refers to a dynamic set of individuals or institutions defined around a set of resource-sharing rules and conditions. All these virtual organizations share some commonality among them, including common concerns and requirements, but may vary in size, scope and structure **[51].**

1.2 Fault Tolerance

Fault tolerance is the ability of an application to continue and completes its execution after the application or part of it, fails. There are several ways available to provide fault tolerance **[52]**. One option to solve failures would be to enable a failed application to return to a previous consistent state and then re-executes from that point. For example, when the current values of a failed application are lost, the object data may return (rollback) to previous values, and processes may return to a state in which a message is re-sent, if the previous attempt apparently failed. **[4]**

It is more difficult to achieve fault tolerance in distributed applications because the applications are made up of several processes that communicate by passing messages between themselves. One process can fail without the other processes noticing the failure thus making the whole application inconsistent in many cases.

1.3 Problem Statement

While collaborating and sharing different resources in several virtual organizations, which is one of Grid computing main objectives, is achievable today, the current Grid standards do not provide relevant information to understand fault tolerance in the environment. Fault tolerance is a major challenging research area and one of the research aims in this area is to enable Grid services to successfully execute long running applications on a Grid. To achieve this, it is important to define efficient fault tolerant mechanisms to ensure the smooth and successful execution of the parallel applications. At the Centre of Parallel Computing (CPC), University of Westminster, PVM (Parallel Virtual Machine) and MPI (Message Passing Interface) based applications are executed on the Grid and therefore it is crucial to ensure that these applications executed reliably. Presently there is no fault tolerant mechanism used by the research centre. This makes applications execution quite unreliable mainly when they run for long hours or even days. A simple failure, be it at hardware, software or network level will require the re-execution of the whole application. This implies losing hours or days of execution. So, to resolve this problem, finding an appropriate fault tolerance solution to execute our parallel applications is important.

1.4 Research Challenges

Though there are several fault tolerant solutions, there is still a lot of research work to be done in this area in order to develop an efficient solution. The main challenge for this research is to find a fault tolerant solution that will enable MPI applications to execute on the Grid. The main focus would be to find a solution that is reliable but at the same time does not affect the execution time of the application in Grid environment too much due to overhead incurred during the checkpointing process. One of the major obstacles in this process is the inter-process communication that exists among the different processes running during the execution of an application. Dealing with the communication layer during the checkpointing process incurs a lot of overhead thus affecting the execution time of the application. Another source of overhead is where the checkpoint files should be stored. The location of the stable storage has a bearing on the overall time in checkpointing and restarting an application.

1.5 Research Objectives

The first objective of this research was to investigate existing fault tolerant solutions in Grid environment and to give critical evaluations where appropriate. The second one is to develop, implement and test a novel fault tolerant solution for executing parallel applications in the Grid environment. The solution will be implemented at the application level and will provide an efficient fault tolerant mechanism that will ensure MPI applications recovers reliably during failures without hugely affecting the performance.

1.6 Research Contributions

The main contribution of this research is the development of a novel approach to checkpoint MPI applications in the Grid environment. A coordinated checkpointing algorithm has been developed that reliably execute MPI applications on the Grid. The solution successfully checkpointed applications at the best possible place and time to minimize checkpointing overhead and hence forth improve performance as compared to other traditional checkpointing solutions that exist.

1.7 Structure of the Thesis

Chapter two introduces the Grid and the different protocols supported by the Grid system. It also analyses the different fault tolerance mechanisms that exist.

The third chapter gives an analysis of the different checkpointing techniques and algorithms that existing before analysing a few projects that use checkpointing as a fault tolerance mechanism.

Chapter four explains the proposed solution and provides the algorithms for the proposed checkpoint/restart process.

Chapter five explains how the proposed solution has been implemented.

Chapter six describes the test bed for the proposed solution. It also gives the tests results for all the tests carried out before analysing these results.

Chapter seven gives a summary of the proposed solution and proposes a few future works that can further improve the implemented solution.

Chapter 2

2 The Grid Systems and Fault Tolerance.

2.1 The Grid Systems.

The first generation Grid systems enabled users to access Grid resources on demand. The main components of the Grid systems were the resource providers, resource consumers and a resource information system. As said earlier, the Grid consists of a collection of resources. However, to form the Grid, the collection of resources requires middleware systems that hide all the details of available resources. The Grid system functionalities are normally organized at several hierarchical levels and the layers are as follows: The fabric layer is the lowest layer and at this level, resources are connected by the network and governed by local resource managers, such as Condor. The Condor Grid middleware system provides a job queuing mechanism, scheduling policy, resource monitoring, priority scheme, and resource management. Users can submit their jobs (serial jobs or parallel jobs) to Condor which places them into a queue. The job manager decides when and where to run the jobs, monitors their progress, and eventually informs the user when the job execution is complete.The second layer contains the core Grid middleware services (e.g. security services, global scheduling, and resource co-allocation). The third layer is the Grid support layer and at this level, higher-level services are built on top of the base services of the second level. (E.g. resource brokers, workflow managers). The fourth layer provides the application services (e.g. Grid portal, analysis and visualization tools) **[1].**

To create such a layered middleware, the first generation Grid systems adopted the Globus middleware; Globus Toolkit 1 (GT1) and Toolkit 2 (GT2)**.** Thus, through the use

of Globus Toolkits, it was easier to create usable Grids, enabling high-speed coupling of computers, databases and instruments. It also allowed users to run a job on more than one machine at the same time, even if these machines are located at geographically different regions and owned by different organizations **[11].**

The second generation of Grid systems combines the Web Services technology with the Grid concept. Web services are a distributed computing technology that allows the creation of applications based on the client/server model. Web services are platform and language independent and they use open and known protocols, such as the HyperText Transport Protocol (HTTP), Web Services Description Language (WSDL) and Simple Object Access Protocol **(**SOAP). The Web services architecture is based upon interactions among three components: Web services provider, Web services registry (Universal Description, Discovery, and Integration - UDDI) and Web services client **[53]**. Web services are non-transient and stateless services. They are called stateless services because these services do not maintain state information between message calls. All information that is required by a Web service has to be stored either in a persistent storage or is passed to it by the caller. This implies that Web services cannot remember result from one invocation to another. This is a problem for the Grid environment where often a set of related computations are executed rather than a single computation. Web services can only execute a set of related computation if they pass the result from one computation to the other as a parameter. Web Services are called non-transient services because they outlive all their clients. This implies that, a Web service still holds the result of the computation executed by the last client. Therefore, a new client accessing the Web service can access the result of the previous user **[14].**

So, a Web service is not a very efficient solution because in distributed systems, there are some situations when transient and stateful services are required. Grid Services can solve this problem and this is one of the main reasons for extending Web services definition by the Open Grid Services Architecture (OGSA) and Open Grid Services Infrastructure (OGSI). Grid services could solve Web services problems, incorporating the concepts of well-defined interfaces, stateful Web services, inheritance of Web services interfaces, asynchronous notification of state change, references to instances of services, collections

of service instances and service state data that augments the constraint capabilities of Extensible Markup Language (XML) Schema. Thus, Web services are the foundation for Grid services, which are the basis of OGSA, OGSI, and therefore, GT3 **[2] [54].**
The aim of OGSA is to define a new common and standard architecture for Grid-based applications. It defines what Grid services are, what they are capable of and what types of technologies they should be based on **[6].** The development of OGSA represents an evolution of the Globus Toolkit 2.0, in which the key concepts of factory, registry, reliable and secure invocation among others, exist, but in a less general and flexible form than Globus Toolkit 3, and without the benefits of a uniform interface definition language **[15].** OGSI is a formal and technical specification of the concepts described in OGSA, including Grid Services **[17] [18].**

However, OGSI was created with the view that it will eventually be converged with Web services standards, making Web services and Grid services become the same thing. Unfortunately, this did not happen because the convergence was unsuccessful **[7]**. This was mainly because the OGSI Grid services have several drawbacks as listed below:

- OGSI is a complex specification with too many standards in it and OGSI developers do not currently use most of them **[21].**
- OGSI does not work well with present Web services and XML tooling. For example, OGSI uses XML Schema with substantial use of xsd:any, attributes, etc. These features cause problems with, for example, JAX-RPC **[21].**
- OGSI's concepts are very object oriented. Even though lots of Web Services systems have object-oriented implementations, Web services themselves are not supposed to be object-oriented. But OGSI takes many concepts from object oriented paradigm (such as statefulness, the factory/instance model, etc.)

To solve this problem, the Web Services Resource Framework (WSRF) has been created. This standard replaces OGSI and has made it easier to converge Grid services with Web services. (See Figure 2.1(a)). This is the third generation Grid system.

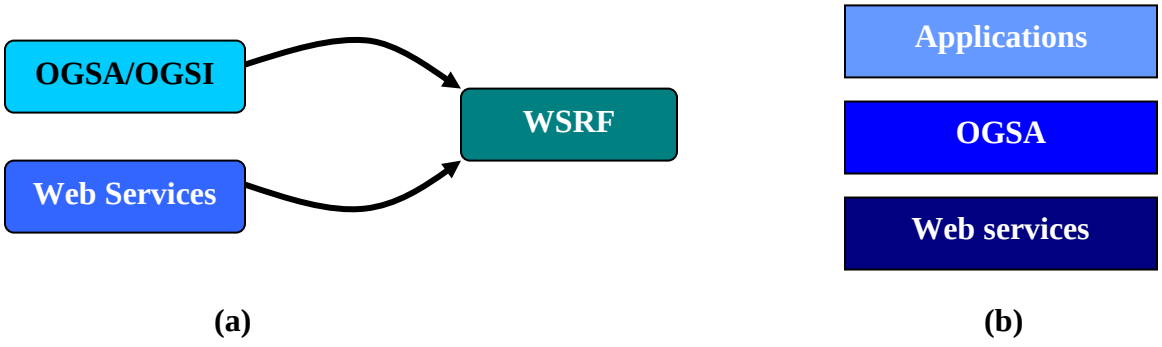

Figure 2.1: Web Services Resource Framework (WSRF)

Unlike OGSI which was a "patch over Web services", WSRF integrates itself into the Web services standards, as shown in figure 2.1(b). OGSA is based directly on Web services instead of being based on OGSI Grid services **[10].** The WSRF makes convergence with Web services possible by solving OGSI's drawbacks. The solutions to the three drawbacks listed above are:

- The OGSI functionality has been split into six independent specifications (instead of one long specification) making it simpler for adoption **[27].**
- WSRF uses standard XML Schema mechanisms that are familiar to developers and are supported by available tooling **[11].**
- Because Web services cannot have state, WSRF separates the service from the state. Support for factory/instance services also disappears, even though design pattern can be used to implement it.

The Globus Toolkit 4 is a WSRF implementation. It provides an API for implementing stateful Web services which target distributed heterogeneous environments **[13].**

2.2 Fault Tolerant Techniques.

Currently, the most common techniques used to manage failures in Grid environment are:

- **Retrying:** If an application crashes, it can be re-executed on a specified Grid resource a certain number of times **[49]**.
- **Replication:** This technique makes replicas of an application that are executed on different Grid resources. As long as not all replicated applications crash, the application will be executed successfully **[49]**.
- **Checkpointing:** When an application is executed, it is checkpointed at various intervals and the checkpoints are stored in a stable storage. If at some point the application fails, the last checkpointed state is retrieved from the stable storage and the application is re-executed from that point rather from the beginning **[49]**.

Retrying is the simplest failure recovery technique to use. If an application failure is detected this application would be retried on a specific Grid resource a certain number of times with a predefined time interval between the tries **[5]**.

In Replication, the application will be replicated on different Grid resources. As long as not all replicated applications crash, the application execution would succeed. When an application is executed, the underlying system simultaneously submits the application execution request to a specific number of Grid resources. Once one of the applications has executed successfully, the replicated applications are stopped **[5].**

Checkpointing is an efficient fault tolerance technique for long running applications in distributed, parallel systems. Presently, there are many checkpoint libraries and program development libraries which support checkpointing. With these checkpointing facilities, checkpoint-enabled applications can be developed by linking these applications with the checkpointing libraries. With checkpointing, when an application fails, it is allowed to be restarted from the recently checkpointed state rather than from the beginning. Checkpointing is a common fault tolerant mechanism to manage failures in the Grid environment as it is very efficient for environments with high failure rates **[5].**

2.3 Existing Fault Tolerant Mechanisms.

In this section, a few existing fault tolerant solutions have been analysed. These solutions have been selected because they are all providing a fault tolerant solution for parallel/distributed applications running on a Grid environment. The solutions cover a range of existing fault tolerance mechanisms such as retrying, replication and checkpointing among others.

The E-Demand project was mainly developed to protect Grid applications against both malicious and erroneous services through the use of a fault tolerant mechanism based on replication. In this project, a coordination service was implemented which uses replication to solve job failures and resources failures. When a job is submitted to a coordinated service(s), the service determines the most appropriate nodes to send the replicas to, and subsequently sends them to these nodes. The nodes then process the job until they complete or until they fail. The coordinated service then evaluates the results of the nodes and randomly selects a node requesting it to send the result back to the client **[12].**

The Fraunhofer failure handling framework was developed to make Grid workflows fault tolerant by providing two types of fault management mechanisms; the Implicit Fault Management and the Explicit Fault Management mechanisms**.** The Implicit Fault Management is included in the Grid middleware and is invoked by lower-level services regarding fault management of each job. The mechanism is based on retrying and is introduced automatically whenever a job fails when being submitted or executed. The Explicit Fault Management mainly refers to the inclusion of user-defined fault management jobs within the Grid workflow. A user-defined fault management job is executed whenever there is a specific failure (disk full, timeout, etc.). For example, in a Grid workflow, we can specify that if a job A is being executed and it does not complete after a specified time (the job fails due to timeout), the job B will be executed. Else job C will be executed. Here all the three jobs are the same except that they are executed on different resources **[17].**

Among the fault tolerant mechanisms that already exist, the Grid Workflow System (Grid-WFS) framework is a promising solution. It is a flexible failure handling framework for the Grid as it supports retrying, replication, checkpointing, replication with checkpointing, alternative task and workflow level redundancy **[9].** Alternative task mechanism makes use of different implementations available for a certain computation (each implementation, however, has different execution characteristics). For example, if there are two implementations A and B, for an application, we can include both implementations in the workflow so that if one application fails (say A), the other application is executed. Here, application B will only start if application A fails. This mechanism is similar to retrying except that instead of retrying the same application, you are executing a different version of the application. Workflow level redundancy mechanism is similar to the alternative task mechanism except that here, the different implementations available for a certain computation are run in parallel. For example, if there are two implementations A and B for a given application, both applications will be executed at the same time on different resources. The result of the application that terminates first is taken. This mechanism is similar to replication except that here you are running different implementation model of the same application in parallel **[19].**

The GriphyN project was developed to generate and manage Grid workflows, ensuring their proper execution in the Grid environment. It uses “abstract” and “concrete” workflow generators to generate workflows automatically. Different abstract and concrete workflow models can be generated for a particular Grid workflow **[3].** The availability of set of different workflow models ensures the successfully execution of a given Grid workflow. For example, if a particular workflow model is being executed and subsequently fails, an alternative model can be adopted an executed. Their fault tolerant solution also includes retrying and replication. The solution also provides the alternative task mechanism which is similar to the one described in the Grid-WFS framework **[31] [32].**

Another fault tolerant mechanism for Grid was developed for P-GRADE. It contains a parallel checkpoint and migration module that enables the checkpoint and migration of

generic PVM programs either inside a Grid site, like a cluster, or among Grid sites when the PVM programs are executed as Condor or Condor-G jobs. To provide fault-tolerance, application-wide checkpoint save is performed, and the checkpoint information is stored on a stable storage server for roll-back if necessary. The checkpointing system is triggered by a grapnel server which is an extra co-ordination process that is part of the application and generated by P-GRADE. When a PVM application is executed, a checkpoint library is loaded at process start-up. This library is activated by receiving a predefined checkpoint signal sent by the grapnel server which then reads the process memory image and passes this information to a Checkpoint Server.

Analysis of existing Fault Tolerant Mechanisms

In this section, the advantages and disadvantages of each of solutions are discussed.

E-Demand

The advantage of this solution is that it does not require any extra software to be supplied from the client application. The same application just needs to be executed concurrently on several Grid resources. However, this fault tolerance solution consumes extra processing power which is a major drawback mainly in dedicated Grids.

Fraunhofer framework

The main advantage of the Fraunhofer framework is its flexibility as it provides a set of fault tolerance solutions. However, it uses only retrying to implicitly solve job failure. This is not the best option because retrying on it own is not effective in the Grid environment where the rate of failures of a job varies a lot. Sometimes the failure rates can be high and sometimes it can be low. When the rate of failure is high, retrying mechanism is not recommended because the expected completion time for a job using this mechanism is very big. Moreover, the mechanism is only concerned with job failure in the workflow. There is no consideration about other types of failures (e.g. node failure, network partitioning) that can in one way or the other affect the proper execution of a Grid workflow**.**

Grid-WFS

The Grid-WFS supports multiple failure recovery techniques as opposed to most other systems described above. It supports retrying, replication, checkpointing, replication with checkpointing, alternative task and workflow level redundancy and looks more promising than the other two solutions. Another advantage of this framework is that the fault tolerant solutions are not hard-coded in application code but are specified using high-level workflow structure which is separated from the application code. The main disadvantage of Grid-WFS framework is its cost benefit. Providing all these fault tolerant mechanisms is an expensive approach. Another disadvantage of this model is that it does not provide any solution for node crashes, network partitioning or any other hardware failures. Workflow level redundancy is not a very efficient solution considering that it will require extra processing power to execute the different jobs in parallel.

GriphyN

The main advantage of GriphyN is that it provides multiple fault tolerant options to provide fault tolerance. The solution does not require any change at the OS level, system level or application level thus making it portable. However, both the concrete and abstract workflow models are based on retrying and replication. Replication, as mention earlier, consumes extra processing power thus affecting performance while retrying is not the best solution in the Grid environment as explained earlier. Moreover, generating various abstract and concrete workflows, requires the re execution of the whole workflow when a failure occurs. This is not a good solution as it could be that only of job in the workflow has caused the failure. So re-executing the whole workflow will be a waste of time and resources.

P-GRADE

The advantage of the P-GRADE framework is that it supports automatic checkpointing of parallel applications. Checkpointed jobs can be migrated to other host in case of host failure. However, P-GRADE can only checkpoint PVM applications. It cannot checkpoint MPI applications Moreover, the checkpointing algorithm used does not consider any options to reduce overheads during checkpointing (e.g. inter process

communication, size of data checkpointed, checkpoint intervals, etc). This means that the checkpointing process can have a negative impact on the execution time of applications.

2.4 Chapter Summary and Conclusions

This chapter gives a brief description of the Grid systems and its evolution before analysing a few fault tolerance solutions that have been used in different projects. The table 2.1 below gives a summary of the fault tolerant mechanism(s) adopted by each solution described in section 2.3.

	Re-Trying	Replication	Checkpointing
E-Demand		X	
Fraunhofer	X		
Grid-WFS	X	X	X
GriphyN	X	X	
P-GRADE			X

Table 2.1: Fault tolerant mechanisms for existing solutions

In this chapter we saw three main fault tolerance mechanisms; retrying, replication and checkpointing. All the three solutions have proved to be successful in the Grid environment though each one has their own advantages and disadvantages.

For this research, the checkpointing solution was chosen for the following reasons:

1. The solution that will be implemented will mainly be used for long running parallel applications and checkpointing has proved to be very efficient for long running applications.
2. The application will be executed on a dedicated cluster and replication will affect performance as it will consume too many resources.
3. Grid environment is generic, heterogeneous, and dynamic with lots of unreliable resources. This is not ideal for long running applications which use retrying as a fault tolerance solution. Each time there is a failure, the application will need to

be re-executed from the beginning. This will definitely affect the execution time of the application. On the other hand, checkpointing is more suited because even though there are lots of unreliable resources in the Grid, applications do not need to restart from the beginning when a failure occurs. This will significantly improve the execution time as compared to retrying.

Chapter 3

3 Checkpointing

3.1 Introduction

Having analysed the different fault tolerant methods, we concluded that checkpointing is best suited for the Grid environment. In the main part of this chapter, different checkpointing techniques were explored and before deciding which techniques is best suited for this research work.

3.2 State of the Art

Today, checkpointing is often used to implement rollback-recovery in distributed environment. This tends to be deployed in large-scale parallel processing environments, such as those used in high-performance computing. Application based checkpointing has been the most commonly used mechanism in such environment mainly for applications that run for hours or days. The checkpointing interval is usually measured in minutes (e.g. every half an hour), and therefore the performance impact of checkpointing is lessened over the long time it takes to run the application **[24].**

3.3 The Different Levels of Checkpointing

Checkpointing is a process where a program's state is saved, usually to stable storage, so that it may be reconstructed later in time. Checkpointing provides the backbone for fault tolerance and process migration among others **[16].**

3.3.1 Checkpointing Techniques

There are three levels where checkpointing can be implemented and they differ in the level of user/programmer involvement. As shown in Figure 3.1, these levels are:

- OS checkpointing.
- Compiler based checkpointing.
- Application based checkpointing.

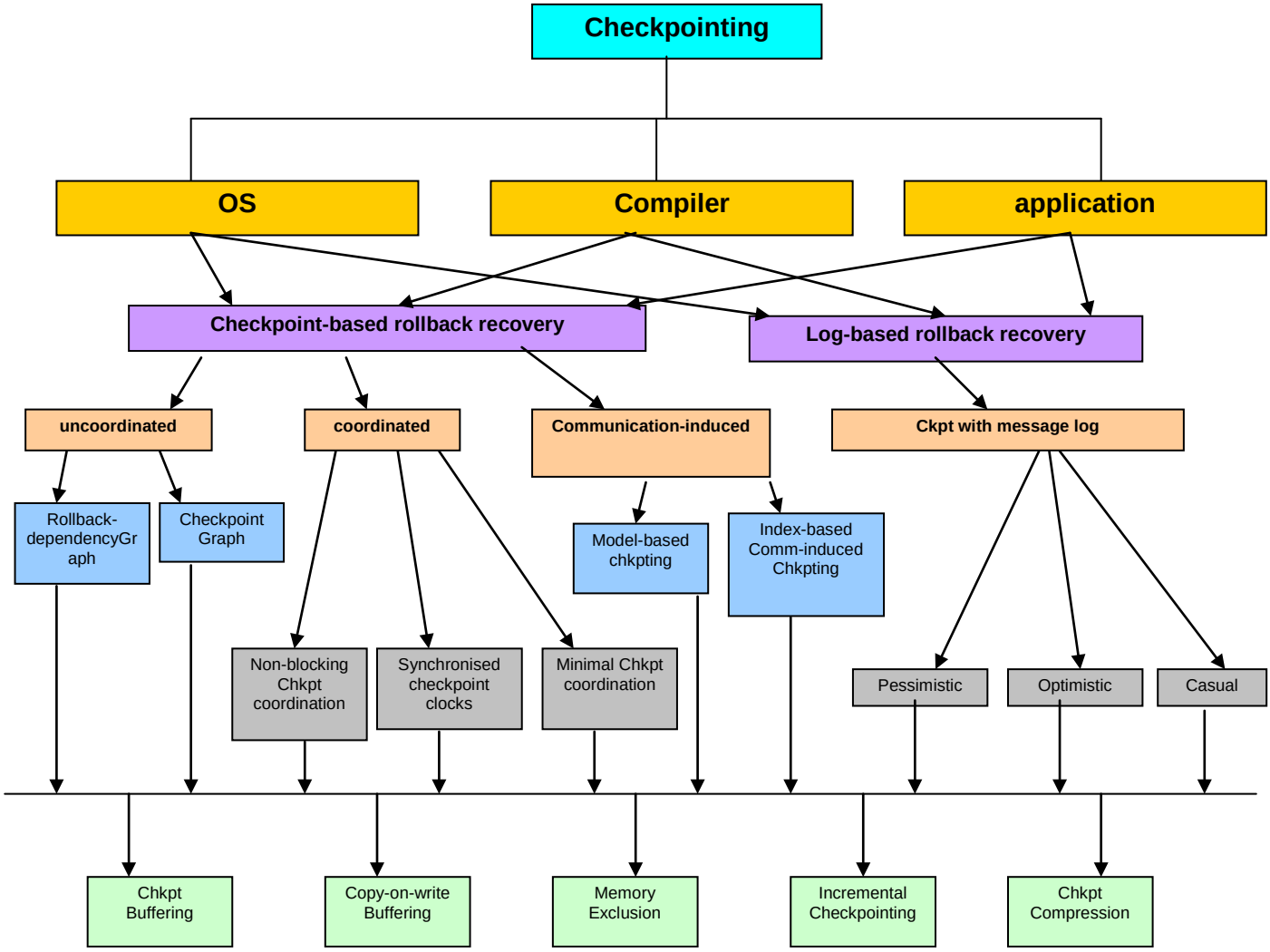

Figure 3.1: Checkpointing Hierarchy

OS Checkpointing

Some operating systems provide checkpointing support inside the operating system kernel without any effort on the part of the programmer or user. The underlying operating system provides automatic recovery. The checkpoints can be triggered on a periodic basis, or the user program may issue a system call to initiate the checkpoint.

The main advantages of this technique are: The programmer is released from consideration of faults. The programmer only needs to specify the interval between checkpoints and the checkpointing process is transparent.

Some of its disadvantages are: Because the checkpointing mechanism is implemented at the system-level, it knows nothing about the semantics of the application. Only the exchanging of messages and the corresponding send/receive events are relevant. The application is seen as a “black-box” and the checkpointing scheme has no knowledge about its internal characteristics. As a result, system-level transparent checkpointing mechanisms typically take gross measures, such as logging all messages or backing up all the processes **[27].**

Compiler Based Checkpointing

In this technique, support for checkpointing is provided inside the compiler. It uses static program analysis to assist the optimisation of checkpointing **[50].** Here, checkpointing is performed at regular intervals by the program itself. To achieve transparency, the program is complied with a checkpointing library. The compiler performs data and control flow analysis on the program, inserting checkpoints at regions where the amount of data to be saved is small **[23].**

The advantage of this technique is that, the placement of checkpoints is transparent to the programmer. The idea is to exploit the knowledge of the compiler to insert the checkpoints at the best places possible and to exclude some inappropriate areas of memory so as to reduce the size of the checkpoint file. Just like Operating System checkpointing, it operates transparently without requiring the programmer to modify the application to insert checkpoint and recovery code **[25] [27].**

Although it can be an effective technique it lacks portability since not all the compilers will include support for checkpointing. It will also be difficult to use in

parallel/distributed applications that communicate through message-passing since the compiler cannot determine the state of the communication channels at the time of the checkpoint. **[6].**

Application Based Checkpointing.

The main method to provide support for checkpointing in this technique is through a run-time library. This approach is not transparent to the user. The checkpoint contents and the places where checkpoints should be taken have to be defined by the application program.
The programmer chooses points within the execution of the application such that the collections of checkpoints taken by all processes will produce global consistent checkpoints. Additionally, the programmer needs to implement the checkpointing and recovery code, including the decision of which data structures to store on stable storage at each checkpoint. The recovery code reads the stored data structures in the checkpoints and reconstructs the connections among the processes in the application **[25].**
This checkpointing technique is excellent for checkpointing parallel/distributed applications in a Grid environment as it is portable. The programmer can specify exactly which data should be saved in a checkpoint operation thus considerably reducing the size of the checkpoint, and consequently, reducing the performance overhead of the checkpoint operation. The programmer has flexibility in controlling the rate of checkpointing.
The principal disadvantage of this technique is the involvement of the programmer. Requiring the application to include checkpointing and recovery code reduces productivity and increases the programmer burden. Making wrong decision about what to checkpoint can result in checkpointing or recovering code that may prevent recovery **[25].**

3.4 Best Approach for This Research

So far, the application based checkpointing mechanism looks most promising for this research. This is mainly because on a Grid environment, portability is very important. Parallel and distributed programs involve a lot of inter-process communication and therefore a compiler based checkpointing mechanism will not be appropriate. Furthermore, to improve performance, it is important to reduce the size of the checkpointing file as much as possible and OS checkpointing techniques do not allow this.

Though application based checkpointing can be a bit tedious, a good programmer can easily implement the solution and benefit the various advantages of this technique.

3.5 Terminology

This section explains a few terminologies commonly used in checkpointing. Through the thesis, these terms are mentioned on various occasions. Without proper knowledge of these terms, it will be difficult to understand the different steps involved in the checkpointing process.

Lost and Orphan Message

Figure 3.2 illustrates two processes whose local checkpoints do not form a consistent checkpoint.

Message m_1 from p_1 to p_2 is a lost message. A message is called a lost message when it is registered as being sent by a process but is not received by any other process. In this scenario we can see that the local checkpoint for p1 shows that a message was sent to p2 but the local checkpoint of p2 does not show receipt of any message m1 from p1. Lost messages may occur when in-transit messages between two processes are not captured by a checkpointing mechanism. Therefore when these two checkpoint files are restored for the application to continue, p2 will never receive the message m1 (unless retransmitted) and this can lead to a failure **[45].**

Message m_2 from p_1 to p_2 is an orphan message. A message is called an orphan message when it is registered as being received by some processes but there is no information as to

which process sent that message. In Figure 3.2 below, the local checkpoint of P2 has received a message from P1. However, the local checkpoint of P1 has no records of having sent a message m2 to P2. When there is a failure and we restart the application, it would be a situation where P2 had received a message that P1 had not yet sent. This is an impossible situation in a failure-free execution of the application. **[22].**

Figure 3.2. Lost and Orphan messages

In-Transit Messages

A lost message is also called and in-transit message. A message is in-transit with respect to a global state if it is recorded as being sent by a given process but not as being received by other process. Checkpointing algorithms have to log such in-transit messages in order to restore the state of channels when a computation has to be resumed from a consistent global state after a failure has occurred. Coordinated checkpointing algorithms can log in-transit messages on stable storage. Some algorithms may wait until all in-transit messages have been delivered before a checkpoint can be taken **[24] [26].**

Figure 3.3 shows an application in progress. Every time, we need to take a global consistent checkpoint, we have to make sure that there are no inconsistencies. In the figure below, *Ckpt1* is an inconsistent checkpoint and *Ckpt2* is a consistent checkpoint. (In this section, I used black bars to represent a local checkpoint). *Ckpt1* is an inconsistent checkpoint because we have an orphan message m1. *Ckpt2* is a consistent checkpoint as both m5 and m6 are in-transit (lost) messages. As explained above, in-transit messages need to be logged to ensure consistency.

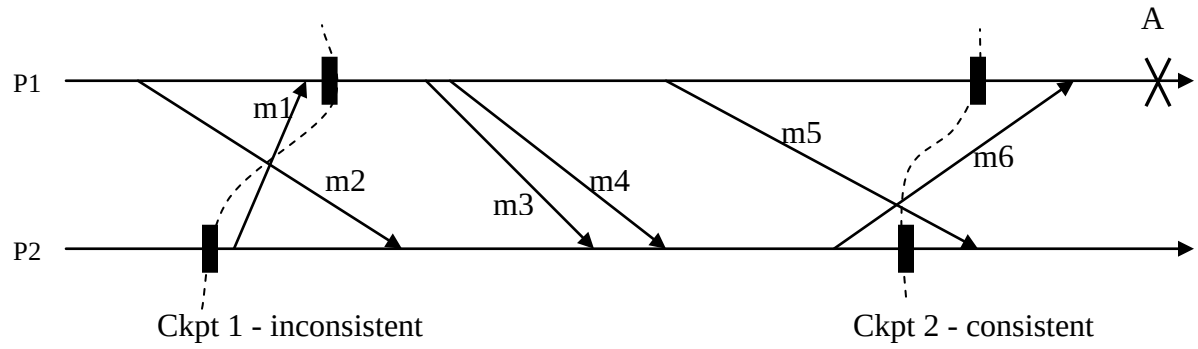

Figure 3.3: Process time diagram

Domino Effect

As said earlier, in a message-passing distributed system, messages induce inter-process dependencies during failure-free operation. Upon a failure of one or more processes these dependencies may force some of the processes that did not fail to roll back, creating a rollback propagation. **[45].**

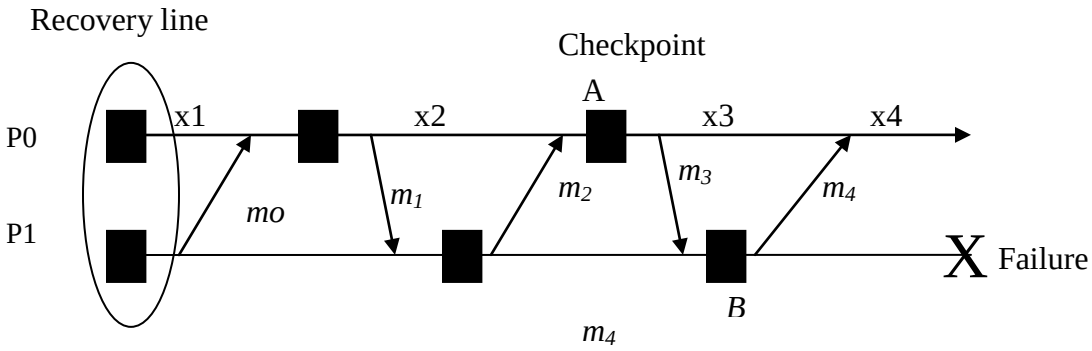

Figure 3.4. Domino effect due to lost or orphan messages

Consider the Figure 3.4 above. It shows an execution in which processes take their checkpoints without coordinating with each other. Each process starts its execution with an initial checkpoint. Suppose process *P1* fails and rolls back to checkpoint *B*. The rollback invalidates the sending of message m_4, putting process *P0* in an inconsistent state. Therefore *P0* must roll back to checkpoint *A* to invalidate the receipt of message

m_4. This cascaded rollback may continue and eventually may lead to the domino effect, which causes the system to roll back to the beginning of the computation, in spite of all the saved checkpoints. In our example shown above, the cascading rollbacks due to the failure of process *P1* may result in a recovery line that consists of the initial set of checkpoints, effectively causing the loss of all the work done by all processes. To prevent the domino effect, processes need to coordinate their checkpoints so that the recovery line is advanced as new checkpoints are taken. **[24].**

Consistent and Inconsistent States

Figure 3.5 below shows two examples of global state. In Figure 3.5 (a) we have a consistent state and in Figure 3.5 (b) we have an inconsistent state.

In Figure 3.5 (a), we have a consistent global checkpoint because there are no lost or orphan messages. Even if message *m1* has been sent but not yet received, the state is consistent because it is a situation in which the message has left the sender and is still traveling across the network. So, during the checkpointing process, the in-transit messages will also be saved as part of the checkpoint process which will be re-transmitted when the application is re-started in case of failure.

On the other hand, we have an inconsistent global checkpoint in Figure 3.5 (b). This is because process *P2* receives *m2* but the state of process *P1* does not reflect sending it. This gives rise to inconsistency resulting to a failure when the application is restarted from the checkpoint state **[30].**

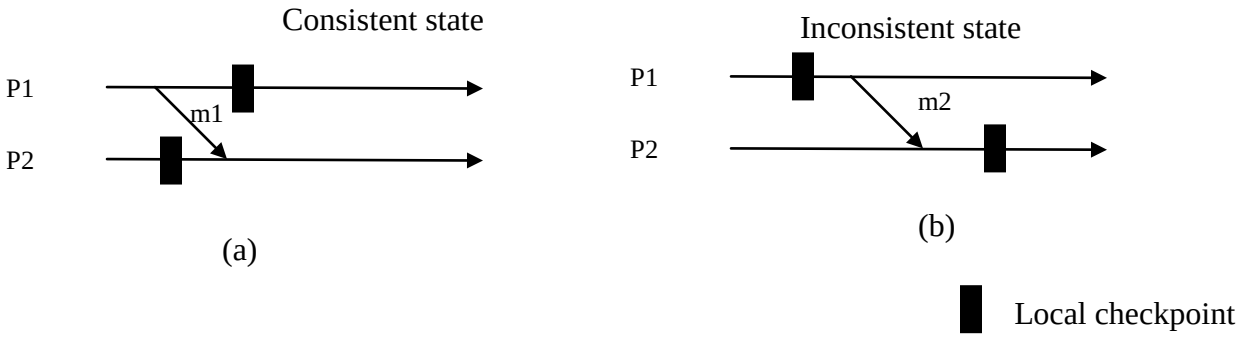

Figure 3.5: A consistent and an inconsistent state

Non-deterministic Events and Deterministic Intervals

To understand log-based rollback recovery, we must first understand what a nondeterministic event and a deterministic state interval is. In log-based rollback recovery, a process execution can be modelled as a sequence of deterministic state intervals, each starting with the execution of a nondeterministic event **[45].** (Log-based checkpointing is explained in the next section).

Non-deterministic events. A process execution is a sequence of state intervals, each started by an event. This event is called a non-deterministic event. Examples of nondeterministic events include receiving messages, receiving input from the outside world, or undergoing an internal state transfer within a process based on some nondeterministic action such as the receipt of an interrupt. For example, in Figure 3.4 above, the execution of process *P0* has 3 non-deterministic events; the receipt of message m_0, m_3, m_4 **[32] [45].**

Deterministic intervals. It is the interval between successive non-deterministic events. In Figure 3.4 above, the execution of process *P0* is a sequence of four deterministic intervals x1, x2, x3 and x4 **[45].**

3.6 Different Approaches for Checkpointing and Recovery

There are many approaches that exist and that can be adopted to implement checkpointing. These approaches, as shown in Figure 3.1, above can be classified into two groups.

1. Checkpoint-based rollback recovery **[30].**
2. Log-based rollback recovery **[30].**

Checkpoint-Based Rollback Recovery

Checkpoint-based rollback recovery relies only on checkpoints to achieve fault-tolerance. When an application fails, checkpoint-based rollback recovery restores the system state to the most recent consistent set of checkpoints.

The main advantage of this approach is that checkpoint-based protocols are simpler to implement than log-based rollback recovery because they do not need to detect, log, or replay nondeterministic events.

But the disadvantage of checkpoint-based rollback recovery is that it does not guarantee that pre failure execution can be deterministically regenerated after a rollback. Therefore, checkpoint-based rollback recovery is not suited for applications that require frequent interactions with the outside world, since such interactions require that the observable behaviour of the system during recovery be the same as during failure-free operation **[31].**

Log-based Rollback Recovery

Log-based rollback recovery combines checkpointing with logging of nondeterministic events. The information needed to replay each event during recovery is logged in each event's determinant. A determinant is a file that contains all the necessary information necessary to replay an event during recovery **[36].** Therefore, by logging and replaying the nondeterministic events, a process can deterministically recreate its pre-failure state.

Consider the example in Figure 3.6 where an execution in which the only non-deterministic events are message deliveries:

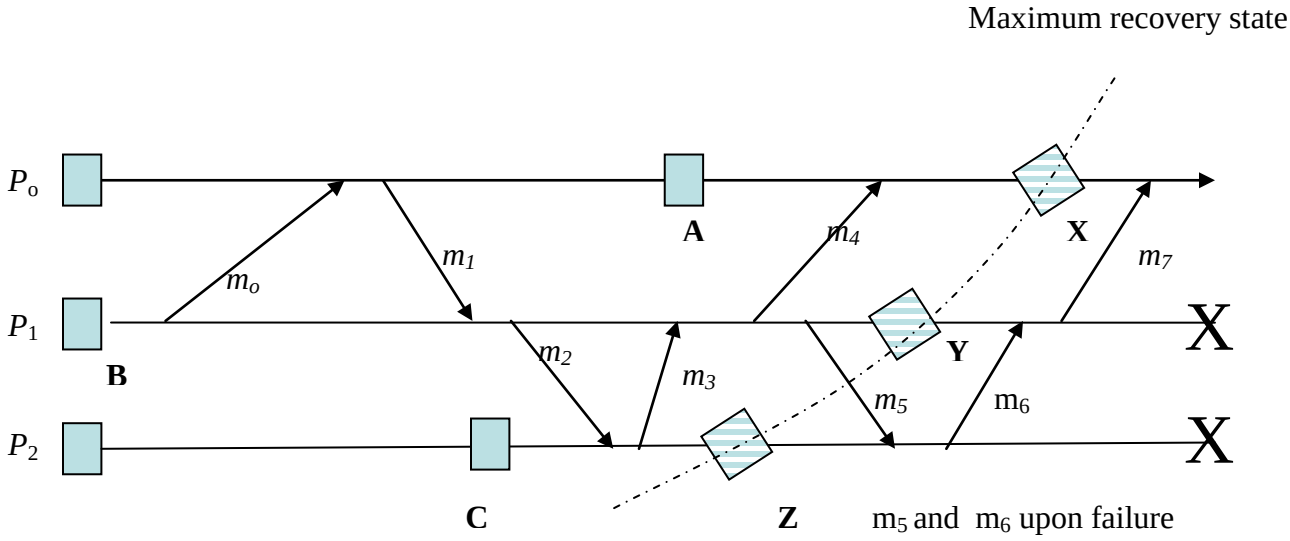

Figure 3.6: Message logging for deterministic replay **[30].**

Assume that processes P_1 and P_2 fail before logging the determinants corresponding to the deliveries of m_6 and m_5, respectively, while all other determinants survive the failure. So, message m_7 becomes an orphan message because process P_2 cannot regenerate m_6 during recovery and P_1 cannot regenerate m_7 without m_6. This makes process P_0 an orphan process and thus force it to roll back as well. Processes P_0 rollback to checkpoint A and replay the delivery of message m_4 to reach state X while process P_2 roll back to checkpoint *C* and replay the delivery of message m_2 to reach state *Z*. Process P_1 rolls back to checkpoint *B* and replays the deliveries of m_1 and m_3 respectively to reach state *Y*. States *X*, *Y* and *Z* form the maximum recoverable state, that is, the most recent recoverable consistent system state **[30].**

During recovery, log-based rollback-recovery protocols force the execution of the system to be identical to the one that occurred before the failure, up to the maximum recoverable state. Therefore, the system always recovers to a state that is consistent with the input and output interactions that occurred up to the maximum recoverable state **[45].**

The main advantage of log-based rollback recovery is that it enables a system to recover beyond the most recent set of consistent checkpoints. This approach is suitable for applications that interact with the outside world which consists of all input and output devices that cannot roll back **[45].**

The main disadvantage of this method it that the checkpointing process can results in a performance overhead due the logging process. Overhead is induced mainly because each message must be copied to the local memory of the process. This may directly affect communication throughput and latency. Another source of overhead is due to the volatile log that is regularly flushed on stable storage to free up space. Another disadvantage would be the size of the checkpoint files due to the logging of non-deterministic events **[29]**.

For this research, the best option would be the checkpoint-based rollback recovery approach. This is because the main focus would be on providing a solution that does not incur too much checkpointing overhead during the checkpointing process while at the same time ensuring that the size of the checkpoint file are kept to a minimal size. Logging non deterministic events will definitely slow down the checkpointing process and increase the size of the checkpointing files.

3.7 Categories of Checkpoint-Based Rollback Recovery

The three most common checkpoint-based rollback recovery categories are:

1. Uncoordinated Checkpointing.
2. Coordinated Checkpointing.
3. Communication-induced Checkpointing.

Uncoordinated Checkpointing

In uncoordinated checkpointing, each process is allowed to take its checkpoints independently, regardless of the dependencies that exist among processes due to message passing. To determine a consistent global checkpoint during recovery, processes record

the dependencies among their checkpoints during failure-free operation. Consider the Figure 3.7 below:

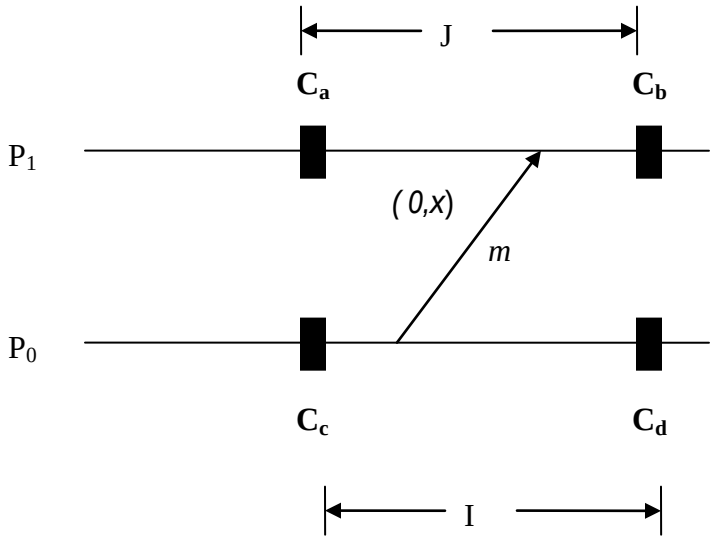

Figure 3.7: Checkpoint index and checkpoint interval.

Here in Figure 3.7,

x = the checkpoint index.

I and J = the checkpoint interval between two successive checkpoints.

If during interval I, process P_0 sends a message *m* to process P_1, it will piggyback the pair (0,*x*) on *m*. Therefore, when P_1 receives *m* during interval J, it records the dependency from I to J which will later be saved onto the stable storage when P_1 takes checkpoint C_b.

If a failure occurs, the recovering process initiates rollback by broadcasting a dependency request message to collect all the dependency information maintained by each process. Based on the global dependency information collected, the initiator calculates the recovery line and broadcasts a rollback request message containing the recovery line. Upon receiving this message, a process whose current state belongs to the recovery line simply resumes execution, otherwise it rolls back to an earlier checkpoint as indicated by the recovery line **[26].**

The main advantage of uncoordinated checkpointing is that each process can take a checkpoint when it is suitable. For example, a process may decide to take checkpoints where the amount of state information to be saved is small, thus reducing overhead. This means that failure free performance overhead is low compared to other checkpoint based recovery techniques **[45].**

There are quite a few disadvantages with this uncoordinated checkpointing. One of the biggest problems of this method is that there is the possibility of the domino effect. This will cause the lost of a large amount of useful work mainly when the processes need to rollback back to the beginning of the computation due to no recovery line. Another problem could be that we may end up with a lot of useless checkpoints that will never be part of global consistent states. Taking unnecessary checkpoints affects performance as it incurs checkpointing overhead and do not contribute in advancing the recovery line **[32].**

Coordinated Checkpointing

Coordinated checkpointing requires processes to coordinate their checkpoints in order to form a consistent global state. When the checkpoint process is triggered, all communications are blocked. One of the processes called the coordinator process takes a checkpoint and then broadcasts a checkpoint request message to all the other processes. When a process receives this message, it stops its execution, flushes its communication channels, takes a tentative checkpoint and sends an acknowledgement back to the coordinator process. When the coordinator process has received an acknowledgement from all the other processes, it broadcasts a commit message to all the processes. Upon receiving the commit message, every process makes the tentative checkpoint permanent. The process is then free to resume execution and exchange messages with other processes **[45].** In Figure 3.8 below there are three processes running, exchanging messages along the way.

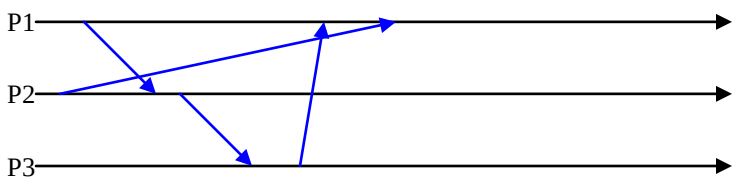

Figure 3.8: Parallel application with three processes

When we need to take a checkpointing, each process will receive a checkpoint request. The processes stop their execution and then sent notification messages to all the other processes (represented by the blue dashed arrows in Figure 3.9) that they are ready to take a checkpointing. Only when all the processes have notify each other and all the in-transit messages have been delivery that a checkpoint can be taken.

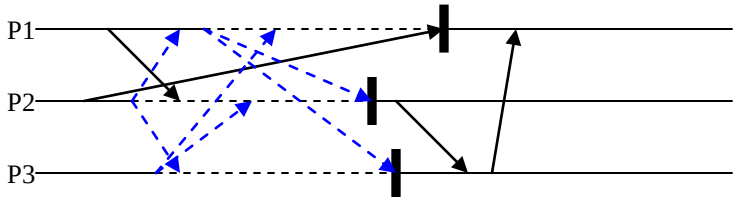

Figure 3.9: Globally coordinated checkpoint

The main advantage of coordinated checkpointing is that the recovery process is simple as it is not prone to the domino effect because each process always restarts from its most recent checkpoint. Unlike uncoordinated checkpointing, this approach requires each process to maintain only one permanent checkpoint on stable storage, reducing storage overhead and eliminating the need for garbage collection **[27].**

The main disadvantage of this approach is that every process has to block for the entire duration until the checkpointing process has completed. Large overhead is involved during the broadcast of checkpoint request message and commit message **[27] [35].**

Communication-Induced Checkpointing

In this approach, the checkpointing process avoids domino effect while allowing some of their process to take some of their checkpoints independently. The checkpoints that a process takes independently are called local checkpoints, while those that a process is forced to take are called forced checkpoints. Forced checkpoints are taken to prevent creation of useless checkpoints. Requests for forced checkpoints are piggybacked on application messages. The receiver of each application message uses the piggybacked information to determine if it has to take a forced checkpoint to advance the global recovery line. A process takes a forced checkpoint only if past communication and checkpoint patterns lead to creation of useless checkpoints **[8].**

As said, communication-induced checkpointing allows processes to take some of their checkpoints independently while preventing the domino effect. However, forced checkpoint must be taken before the application may process the contents of the message. This may lead to high latency and overhead. Moreover, if a large number of forced checkpoints are taken, it will nullify the benefit accrued from the autonomous local checkpoints **[19].**

Best Category for this Research

The following table gives a summary of the checkpoint-based rollback recovery categories.

	Uncoordinated Checkpointing	**Coordinated Checkpointing**	**Communication Induced Checkpointing**
Garbage collection	No	No	No
Checkpoints per process	Several	1	several
Domino effect?	Possible	No	No
Orphan Messages?	Possible	No	Possible
Rollback extent	Unbounded	Last global checkpoint	Possibly several checkpoint
Complex Recovery	Yes	No	Yes
Output Commit	Not possible	Very slow	Very slow

Table 3.1: A summary of the checkpoint-based rollback recovery categories

Coordinated checkpointing is more suitable mainly because of its simpler design and recovery characteristics. Coordinated checkpoint has the advantage of a low overhead as long as the execution stays fault free. Its drawbacks are the synchronization cost before the checkpoint, the cost of synchronized checkpoint and the restart cost after a fault **[33]**. The performance of coordinated checkpoint is also related to where the global checkpoint file will be stored. If the checkpoint images are stored remotely on an independent

checkpoint server, it will induce high overhead during the checkpointing and restart process, even if the checkpoint system uses several checkpoint servers **[38].** However, most of the overhead problems in coordinated checkpointing can be solved to a reasonable level and this is the aim of this research work.

3.8 Analysing Existing Checkpointing Solutions

In this section, several checkpointing mechanisms implemented in different solutions to ensure reliability of applications have been analysed. A brief overview of each solution is given before its advantages and disadvantages was analysed. The analysis mainly focuses on how the fault tolerance solution provides reliability, flexibility, portability and performance.

3.8.1 CryoPID

CryoPID is a checkpointing package that is used to capture the state of a running process and save it to a checkpoint file. It is supported by the Linux kernel (2.4 and 2.6), Intel 8086 CPU and AMD64. It can stop processes at any time to take a checkpoint before restarting it. It allows for progress migration between machines and between kernels of different versions. CryoPID checkpoints a parent process together with all its associated child processes and deals with the file system. Using CryoPID, you can save the dynamic libraries, open files, sockets and FIFO's associated with the process into the checkpoint. The checkpointing package consists of a program called freeze that captures the state of a running process and writes it into a file. To restart the checkpoint, you simply run it as it is self-executing and self-extracting **[55]**.

Critical Analysis

The main advantages of this checkpointing package are:

- No root privileges are needed to run the checkpoint.
- There is no need to modify the kernel.
- There is no need to recompile/relink the checkpointed application.

The main disadvantage of CryoPID is that is cannot be used to checkpoint multi-process applications.

3.8.2 DMTCP

DMTCP (Distributed MultiThreaded Checkpointing) is a transparent user-level checkpointing package for parallel/distributed applications. It transparently checkpoints applications consisting of many nodes, processes, and threads as well as desktop applications **[56]**. No re-compilation and no re-linking to the applications are required. According to J. Ansel et al. **[57]**, DMTCP can checkpoint user-level, multithreaded, distributed processes connected with sockets. The checkpointing time is fast, with negligible overhead while not checkpointing.

Critical Analysis

The main advantage is that it requires no system privileges to operate thus making DMTCP to easily bundled with the application.

The main disadvantage of DMTCP is that it gives good checkpointing response when running on a small size cluster. However, the checkpointing time and checkpointing overhead associated increases as the number of nodes increases on the cluster. The checkpointing package has not been tested in a Grid environment and therefore we do not know how efficient and reliable it is such environment. Another disadvantage of DMTCP is that it does not yet support Infiniband or Myrinet for OpenMPI **[57]**.

3.8.3 The InteGrade Grid Middleware.

Researchers at the University of Sao Paulo have implemented a checkpoint-based rollback recovery of parallel BSP (Bulk Synchronous Parallel) applications running on the InteGrade middleware.

The checkpointing mechanism periodically saves application state to permit to restart its execution from the last saved global checkpoint in case of failure. A precompiler automatically instruments the source-code of a C/C++ application, adding code for saving and recovering application state. They have also implemented runtime libraries

necessary for the generation of checkpoints, monitoring application execution and node failures, and coordination among processes in BSP applications. A failure detector monitors the application execution. In case of failure, the application is restarted from the last saved global checkpoint.

The checkpointing precompiler saves the execution stack and the heap state which contains important data to be used during recovery. It is also responsible for checkpoint generation coordination.

The runtime libraries provide basic functionality for checkpointing of single processes, and specific functionality for checkpointing BSP applications. They provide a C API that allows applications written in both C and C++ to use them. The basic checkpointing functionality is provided by functions to manipulate the checkpoint stack, to save the stack data to a file, and to recover checkpointing data **[28]**.

Critical Analysis

The main advantage of this model is that it only saves the data necessary to recover the application during the checkpointing process. This reduces the checkpointing and restart overhead.

The main disadvantage of the scheme is that it involves periodically stopping normal execution in order to save a checkpoint. Moreover, they store their checkpointing files only locally and this can be a problem in case the machine where the process was executing becomes unavailable. Another current restriction is that the saved data is architecture dependent. This dependency arises due to differences in data representation and memory alignment. Making the checkpoint portable requires saving data in a platform independent format else lots of time is wasted in converting the saved data into relevant format before restarting the process on another location **[28].**

3.8.4 Integrating Fault-Tolerance Techniques in Grid Applications.

Researchers have developed coordinated checkpointing mechanisms for two algorithms: Single Program Multiple Data applications (SPMD) checkpointing and 2-phase commit distributed checkpointing (2PCDC).

SPMD Checkpointing

SPMD applications are common in Grids. An SPMD application is an application that contains a loop that performs calculations on a subset of the data and exchanges information periodically. Checkpoints are inserted at points in the program to successfully obtain a consistent checkpoint.

The global checkpoint files are stored on a checkpoint server. An application manager controls the creation of objects and is responsible for determining when a checkpoint is consistent. When a checkpoint occurs, each process takes a local checkpoint and forwards the checkpoint file to the checkpoint server. Once the application manager receives a confirmation from each process, it informs the checkpoint server that the checkpoint is consistent. Each process has an interface that consists of functions to save and restore the local state, to notify the manager that it is alive, to notify the manager that a checkpoint has been taken successfully and to determine whether the process is in recovery mode.

As mentioned, the application manager is also responsible for determining if a process is alive or has failed. If a process has failed, the application manager then proceeds to restart the application by killing and restarting each process. The processes then request the necessary state from the checkpoint server and restart **[37].**

"2-Phase Commit" Distributed Checkpointing

2-Phase Commit Distributed Checkpointing (2PCDC), is an algorithm which relieves developers from the burden of establishing consistent checkpoints. The basic idea behind 2PCDC is to produce a consistent application checkpoint atomically. Atomicity ensures that the algorithm can tolerate failures when it is in progress and it also ensures the existence of at least one consistent checkpoint at any given time. The algorithm ensures

that no in-transit messages are lost by capturing in-transit messages. The 2PCDC algorithm proceeds in two phases. In the first phase, a coordinator process requests the other processes to take a checkpoint. A process can either accept or reject the request. To accept the request, the process need to send a “Yes” reply to the coordinator and to reject the request, a process need to send a “No” reply to the coordinator. Each process then waits for the coordinator’s decision. While waiting, each process also forward any in-transit message to the checkpoint server and informs the coordinator that it has received an in-transit message. The coordinator will trigger the checkpointing process only if all processes reply “Yes”. This is the end of the first phase. If the decision is “Yes”, the coordinator informs the checkpoint server that the checkpoint is consistent and sends its decision to all processes. If the decision is “No”, the coordinator informs the checkpoint server to discard the local checkpoints just stored.

To prevent orphan messages, a participant is not allowed to initiate communication with another once it has taken a local checkpoint. The algorithm handles lost messages by including a message count with each participant’s reply. To determine whether all in-transit messages have been caught, the coordinator sums the count from each participant. If the total number of sent messages equals the number of received messages then all in-transit messages have been caught and the set of local checkpoints and in-transit messages form a consistent checkpoint **[37].**

The recovery protocol also proceeds in two phases. During the first phase, the coordinator process sends protocol information to each process. The information sent informs processes that they are in recovery mode. Each process then retrieves its state and the in-transit messages from the checkpoint server and informs the coordinator that it is ready to proceed. The coordinator then awaits the ready notification from each process. In the second phase, the coordinator informs each process to proceed **[37].**

Critical Analysis

The main advantage of these two algorithms is that they work well in a Grid environment. They are coordinated checkpointing mechanisms which ensure that consistent global checkpointing are taken efficiently.

However, both approaches have not considered any ways to improve the checkpointing process so as to reduce checkpointing overhead. Also, they designed these algorithms to cope with permanent host failures. They assumed that a host will fail by crashing and that it will never recover. But this is not always the case and in such situation, it can be a further waste of time trying to look for another host.

They also assumed that the hosts on which the checkpoint servers are located and the host that starts the application do not fail. However, if this failure assumption is violated, that is, the host on which a checkpoint server is located crashes permanently, then the application will cease to be restartable.

3.8.5 P-GRADE: A Grid Programming Environment

P-GRADE provides a high-level graphical environment to develop parallel applications transparently both for parallel systems and the Grid. P-GRADE implements an automatic checkpoint mechanism for parallel programs which supports the migration of parallel jobs inside the workflow providing a fault-tolerant workflow execution mechanism. It contains a parallel checkpoint and migration module that enables the checkpoint and migration of generic PVM programs either inside a Grid site, like a cluster, or among Grid sites when the PVM programs are executed as Condor or Condor-G jobs **[39].**

The Checkpoint Mechanism

Their checkpointing algorithm is based on coordinated, non-blocking checkpointing with no modification of the underlying message passing system necessary. The checkpointing procedure is controlled by a library called the GRAPNEL library. So no modification of the user code or the underlying message passing library is required to support process and application migration. The GRAPNEL Server performs a consistent checkpoint of the whole application. Checkpoint files contain the state of the individual processes including in-transit messages so the whole application can be rebuilt at any time and on the appropriate site. The checkpoint system can migrate PVM processes both inside a cluster and among clusters **[39].**

Critical Analysis

The main advantage of this model is its ability to checkpoint and migrate a PVM application in a grid environment. It is a coordinated, non-blocking checkpointing algorithm that performs checkpointing at regular intervals.

The main problem with the algorithm is that it can only checkpoint PVM applications either running on a cluster, or among Grid sites when the PVM programs are executed as Condor or Condor-G jobs. The mechanism has not emphasised on minimizing checkpointing and restart overhead.

3.8.6 OPEN MPI

MPI stands for the Message Passing Interface **[47]**. MPI is a standardized API typically used for parallel and/or distributed computing. Open MPI is a free, open source MPI implementation. Open MPI is therefore able to combine the expertise, technologies, and resources from all across the High Performance Computing community in order to build the best MPI library available **[48]**. Open MPI provides a stable platform for third party research and commercial development **[31].**

The Open MPI architecture

Open MPI consists of three layers that combine to provide a full featured MPI (See Figure 3.10). Below the user application is the Open MPI (OMPI) layer that presents the application with the expected MPI specified interface. Below that is the Open Run-Time Environment (ORTE) layer. This layer provides a uniform parallel run-time interface regardless of system capabilities. The third layer is the Open Portable Access Layer (OPAL). This layer abstracts the irregularities of a specific system away to provide maximum portability. Below the OPAL layer, is the checkpoint/restart system available for the operating system running on the machine. Open MPI uses the Modular Component Architecture (MCA) to define the OPAL Checkpoint and Restart Service (CRS) framework as a uniform API for checkpoint/restart systems. The OPAL CRS design allows for checkpoint/restart of MPI applications running on heterogeneous systems **[41].**

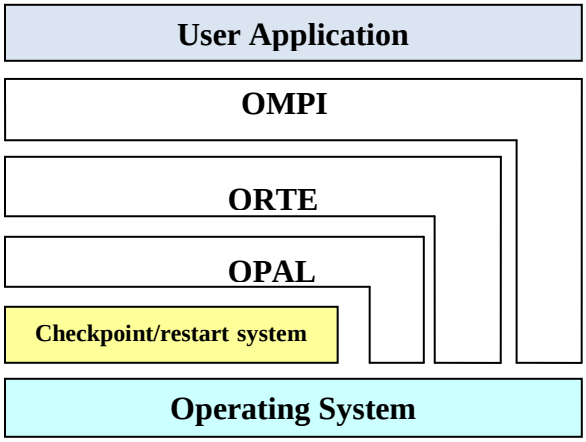

Figure 3.10: The OpenMPI Architecture **[40]**

The Checkpoint/Restart System

The checkpoint/restart system is responsible for preserving and restoring the state of a single process on a single machine. Once the checkpoint/restart system has completed a checkpoint, it must provide Open MPI with a structure containing a reference to the checkpoint image (or images) generated, denoted by the term snapshot reference **[46].**

The Checkpoint/Restart Protocol

Open MPI uses the snapshot references from all of the processes to create a global snapshot of the application during the checkpointing process. The creation of the global snapshot is determined by the checkpoint/restart protocol. By using snapshot references instead of the actual files to create the global checkpoint snapshot, Open MPI can combine snapshot references from different checkpoint/restart systems into a single global snapshot, allowing checkpoints on heterogeneous systems.

Further, by using the global snapshot, Open MPI could arrange for the migration of a single process or the storage of the global snapshot to a remote server, without requiring knowledge of the checkpoint/restart system used or how the checkpoint images have been preserved **[41]**.

The Checkpoint and Restart Service (CRS) Framework

When a process receives a checkpoint, a function called the "Opal Entry Point" function first calls intra-layer coordination callback with the "CHECKPOINT" state indicating that the layers are to prepare for a checkpoint. Once the coordination function has finished, the "Opal Entry Point" function initiates the checkpoint by using the "Opal Crs Checkpoint" function. Through OpenMPI's component system, "CHECKPOINT" invokes the back-end checkpoint/ restart system to begin the process checkpoint. Once the checkpoint is done, the backend checkpoint function return's indicate if execution is to continue or restart **[40]**.

The main functions of the API are:

1. Checkpoint() - The "Checkpoint" function initiates the checkpoint of a single process, identified by its PID, by calling the checkpoint/restart system's checkpoint routines **[40].**
2. Restart() - The "Restart" function initiates the restart of a single process from a snapshot reference by interacting with the checkpoint/restart system's restart functionality**[40].**

Critical Analysis

The OpenMPI in itself is not a checkpointing mechanism. However it provides a solution to enable existing checkpointing mechanism to plug in its architecture in order to provide fault tolerance to MPI applications running in the grid. The main advantage of the solution is its ability to migrate processes without requiring knowledge of the checkpoint/restart system used or how the checkpoint image(s) have been preserved.

As other solutions, OpenMPI needs to coordinate in-transit messages to ensure no lost or orphan messages. This adds to the checkpointing overhead resulting in longer checkpointing and restart time.

3.9 Chapter Summary and Conclusions

In uncoordinated checkpoints protocols without message log, the checkpoints of each process are executed independently of the other processes and no further information is stored on a reliable media leading to the well known domino effect (processes may be forced to rollback up to the execution beginning). Since the cost of a fault is not known and there is a chance for losing the whole execution, these protocols are not used in practice. Communication induced checkpointing tries to take advantage of both uncoordinated and coordinated checkpoint techniques. Based on the uncoordinated approach, it piggy backs causality dependencies in all messages and detects risk of inconsistent state. When such a risk is detected, some processes are forced to checkpoint. In coordinated checkpointing, processes needs to be organised to form a consistent global state. During the checkpoint process, all communications are blocked. A coordinator process manages the checkpoint process by communicating will all the processes in a predefined way to ensure all the communication channels are save together with the process's state.

Most of the projects analysed in section 3.8 described above has adopted the coordinated checkpointing approach as it is most suitable for the Grids environment due to its characteristics. Each solution has its own advantages and disadvantages. The first checkpoint mechanism is a checkpointing package that can be adopted by different architectures to incorporate a checkpointing mechanism in their solutions. CryoPID is an application-level checkpointing mechanism that can easily be adopted by application without the need to re-compile and re-link the application. However, the solution is not suitable for checkpointing multi process applications. The second checkpoint mechanism is also a checkpointing package that can be adopted by different architectures to incorporate a checkpointing mechanism in their solutions. Just like CryoPID, DMTCP does not require system privileges and the checkpointing process is fast. However, the checkpointing time and checkpointing overhead increases as the number of nodes increase thus affecting performance. No approach has been considered to reduce overhead during the checkpointing and restart process. Moreover, DMTCP has not yet been tested in the Grid environment. The third solution provides a good fault tolerance solution allowing the user to decide where to place checkpoint requests and what files to

save. However, the solution does not take into consideration any approach to improve performance. Moreover, the solution is not portable, restricting its use in the Grid environment. The fourth solution uses the coordinated checkpointing process to checkpoint their application. The solution is basic without considering how to improve the performance by looking into issues such the checkpoint intervals, the size of the checkpoint files saved and the location of the saved checkpoint file. The fifth solution works very well for checkpointing PVM application. It also enables migration of PVM applications in case of permanent node failure. However, the solution is specific to PVM applications and also fails to consider criteria to improve the performance of the checkpointing process.

Likewise, Open MPI is a very good solution for checkpointing MPI application but the fault tolerance solution is still at a primitive phase. It does the basic checkpointing of MPI application without looking into methods to decrease overheads incurred during the checkpointing process.

Chapter 4

4 Improving Checkpointing Solutions

4.1 Introduction

In this chapter, the proposed solution is explained. A detailed description about checkpoint interval, First Order Approximation (FOA) and Natural Synchronisation Points (NSP) and eventually the new approach are explained.

Terminology

This section explains a few terminologies commonly used in this chapter.

The First Order Approximation, also referred as FOA is the mathematical derivation implemented by John Young **[43]** to calculate the best checkpointing interval for a particular application.

A forced synchronisation point is a region in a parallel application where synchronisation of processes is required during the checkpointing process. Synchronisation will ensure that no orphan or lost messages occurs during the checkpointing process.

A Critical Region is a user defined region round a predefined regular checkpointing interval.

4.2 Research Proposal

From the previous chapters, we concluded that application based coordinated checkpointing is an appropriate solution for this research because it gives the programmer more flexibility to develop an algorithm that can improve performance. This

checkpointing method gives the programmer the flexibility to control the rate of checkpointing thus enabling him to choose a rate that has minimal performance overhead. The programmer exploits the knowledge about the application to insert checkpoints at the points in the execution where the amount of data to be stored is small. To reduce the cost of checkpoints and thus improve performance, an algorithm was developed that ensures that checkpoints are taken at the best possible checkpoint intervals and the amount of information save being restricted to a minimal. To achieve these goals, the following areas were explored to implement a new algorithm:

- A technique to find the most suitable checkpoint intervals in an application. The First Order Approximation can be used as a starting point to determine these checkpoint intervals.
- Identification of points in the application at which checkpoints will be taken. Points within the execution of the application must be identified such that the collections of checkpoints taken by all processes will yield global consistent checkpoints and the intervals yield improved performance. Performance will improve mainly by exploiting the knowledge of the application to insert checkpoints at the points in the execution where there is minimal overhead.

4.2.1 Natural Synchronization Points

Normally, when a coordinated global checkpoint is taken, messages should be synchronized to ensure that all in-transit messages are preserved. Performance wise, this can be improved by trying to eliminate unnecessary idle time involved during the synchronisation process. When taking a checkpoint, processes are forced to block their computations to perform synchronisation, which degrades performance. Synchronisation requires extra communications among processes. The checkpointing process also has to take into consideration the communication messages, logging in-transit messages or waiting for in-transit messages to be delivered before checkpoints can be taken. Therefore, coordinated checkpointing suffers from high overhead which affects performance a lot **[26].**

The Synchronisation Process

Consider the Figure 4.1 below which illustrates an interaction among three processes. To perform a checkpoint, processes must synchronise the messages to ensure we do not lose any of them. Figures 4.2 and 4.3 illustrate how the checkpoint is taken.

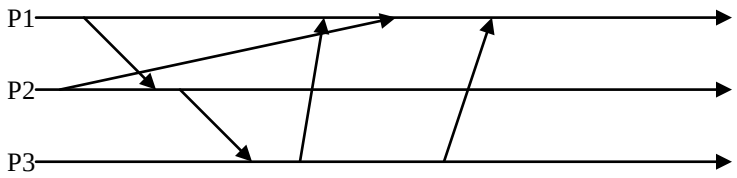

Figure 4.1: Processes interacting

Horizontal dashed lines denote processes idled by synchronization. Dashed arrows represent extra communication that is required for synchronization to happen. Short vertical bars denote the times at which local checkpoints are taken.

In Figure 4.2, processes have to wait for the in-transit messages to be delivered before a checkpoint can be taken. To achieve this, synchronization among processes must occur. The processes stop their execution and then sent notification messages to all the other processes (represented by the dashed green arrows) that they are ready to take a checkpointing. Only when all the processes have notified each other and all the in-transit messages have been delivered that a checkpoint can be taken **[44]**.

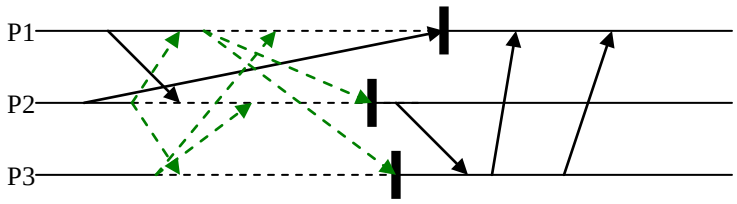

Figure 4.2: Globally coordinated checkpoint – waiting for in-transit messages

To improve the checkpointing process, in-transit messages can be logged rather than wait for them to be delivered. Figure 4.3, is similar to Figure 4.2 except that the synchronisation process is shorter and therefore the checkpointing process happens quicker. On the other hand however, the restarting processes will take longer because

apart from restore the checkpoint files, we need to restore the in-transit messages as well **[44].**

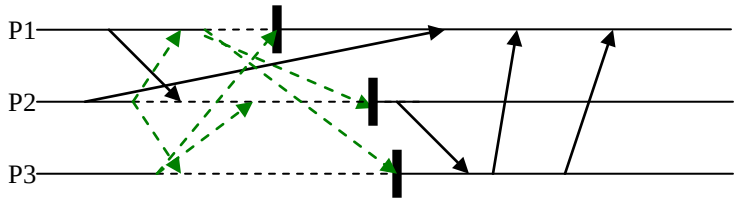

Figure 4.3: Globally coordinated checkpoint – logging in-transit messages

Identifying Natural Synchronisation Points

In parallel programs, the periodic exchange of boundary information establishes natural points for taking application consistent checkpoints, e.g., at the top or the bottom of the main loop. In addition, at the end of iteration the size of the global checkpoint state is often minimal. There are also other existing synchronization points such as barriers and collective operations that represent natural consistent global states. Also, at these points the state of the process stack is useless and therefore need not be stored, thus reducing the size of the checkpoints. These regions are called Natural Synchronization Points. Placing checkpoints in these regions is beneficial as it helps avoid the overhead of forcing a global consistent state to take a global checkpoint. So, programmers can use these natural synchronization points of the application to perform global consistent checkpointing. There is no need to worry about the state of the communication channels or in-transit messages, because there are no inter-process communications at these points **[27].**

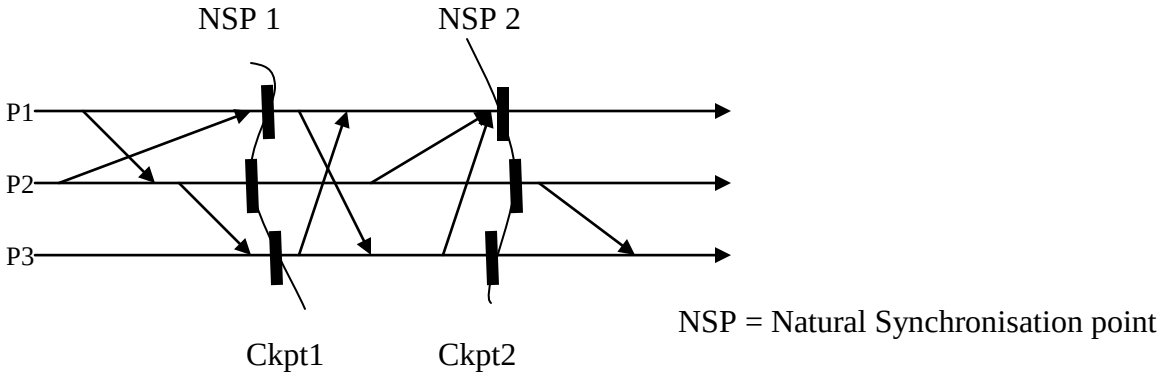

Figure 4.4: Checkpointing at natural synchronisation points

Figure 4.4 above, shows two natural synchronisations points; NSP1 and NSP2. At these points, no messages flow across the natural synchronisation points. So, taking checkpoints at NSP1 and NSP2 guarantee that there will not be any lost or orphan messages thus ensuring consistent checkpointing.

Problem with checkpointing at Natural synchronisation points only

Because there is no pattern in the occurrence of natural synchronisation points, we cannot rely only on these points for checkpointing. This is because we may have long periods between two successive natural synchronisation points and not taking a checkpointing for a long period may affect the efficiency of our checkpointing mechanism.

4.2.2 First Order Approximation

To improve the checkpointing mechanism, we must ensure that checkpoints are not taken at all natural synchronisations points but at intervals that will ensure the best performance. To achieve this, the First Order Approximation Method proposed by John W. Young was analysed. Young developed a mathematical formula based on Maclaurin expansion to calculate more appropriate intervals to perform checkpoints. He named it the “optimal checkpointing interval” **[43].**

Order of Approximation

In science, engineering, and other quantitative disciplines, orders of approximation refer to formal or informal terms for how precise an approximation is. They indicate progressively more refined approximations: in increasing order of precision, a zeroth order approximation, a first order approximation, a second order approximation, and so forth **[58]**.

Scientists use the zeroth-order approximation for a first estimate at an answer. A zeroth-order approximation of a function is a constant value, or a flat line with no slope. That is, a polynomial of degree 0. They use the first-order approximation for a further estimate at an answer. A first-order approximation of a function is a linear approximation, straight line with a slope. That is, a polynomial of degree 1. Scientists use the second order

approximation for a decent-quality answer. A second-order approximation of a function is a quadratic polynomial, geometrically, a parabola. That is, a polynomial of degree 2 **[59]**.

Maclaurin Expansion

Maclaurin expansions can be used to represent certain functions in polynomial forms, which can then be used to provide different order of approximation. A polynomial series is a mathematical expression consisting of added terms, terms which consist of a constant multiplier and one or more variables raised to integral powers. For example, $3x^2 - 2x + 7$ and $5y + 8x^3$ are polynomials **[60]**.

Under certain conditions mathematical functions can be written as polynomial series. For example, the quadratic function $f(x) = (x+1)^2$, can be represented as $f(x) = x^2 + 2x + 1$ which is a polynomial series. This kind of expansion is a special case of the more general Binomial Theorem **[60]**. For example,

$$(1+x)^n = 1 + n\,x + \frac{n\,(n-1)x^2}{2!} + \frac{n\,(n-1)\,(n-2)\,x^3}{3!} + \frac{n\,(n-1)\,(n-2)\,(n-3)\,x^4}{4!} + \ldots \quad (1)$$

Where the '+…' indicates that it is an open-ended polynomial that goes to infinity. The factorial number 2! means 2*1; 3! means 3*2*1; etc **[60]**. For example, for n= 4, the above Binomial Theorem results in $(1+x)^4 = 1+4x+6x^2+4x^3+ x^4$.

Therefore, using the Binomial Expansion we can have polynomial representations of functions of the type $(1 + x)^n$. However, to represent transcendental functions (e.g. sine, log, exponential) as polynomial series, Maclaurin series are used with some assumptions **[60]**.

Consider a function *f(x).* Assume the function can be expanded as:

$$f(x) = a_0 + a_1x + a_2x^2+a_3x^3+a_4x^4+a_5x^5 + a_6x^6+\ldots \quad (2)$$

As we can see, the function has been expanded as a polynomial series. At present, the polynomial series has undetermined coefficients a_0, a_1, a_2, a_3, a_4, a_5, a_6, ...
If we put $x = 0$ in the above series, assuming the function exists at $x = 0$, we get:

$$f(0) = a_0 \quad (3)$$

Next, assuming that the function and its polynomial representation are differentiable, then

$$f'(x) = a_1 + 2a_2x + 3a_3x^2+4a_4x^3+5a_5x^4+6a_6x^5 + \ldots \quad (4)$$

By putting the value of x to be 0 in this series and assuming the differential of the function exists at $x = 0$, we get

$$f'(0) = a_1 \quad or \quad a_1 = f'(0) \quad (5)$$

Next, assuming that the function and its polynomial representation are now twice differentiable, then

$$f''(x) = 2a_2 + 3*2a_3x+4*3a_4x^2+5*4a_5x^3+6*5a_6x^4 + \ldots \quad (6)$$

By putting the value of x to be 0 in this series and assuming the second differential of the function exists at $x = 0$, we get

$$f''(0) = 2*1*a_2 \quad \text{or} \quad a_2 = f''(0) / 2! \quad (7)$$

Similarly,

$$a_3 = f'''(0)/ 3!\ ,\ a_4 = f^{(iv)}(0)/4!\ ,\ a_5 = f^{(v)}(0)/5!\ , \ldots \quad (8)$$

So, with the unknown coefficients found in terms of the value of the function and its derivatives at $x = 0$, the general Maclaurin Theorem of any infinitely differentiable function $f(x)$ can be written as:

$$f(x)= f(0) + f'(0)x + \frac{f''(0)}{2!} x^2 + \frac{f'''(0)}{3!} x^3 + \frac{f^{(iv)}(0)}{4!} x^4 + \frac{f^{(v)}(0)}{5!} x^5 + \frac{f^{(vi)}(0)}{6!} x^6 + \ldots \quad (9)$$

For example, using Maclaurin expansion, the function e^x, can be represented in a polynomial form as follows:

$$e^x = 1 + x + x^2/2! + x^3/3! + x^4/4! + x^5/5! + \ldots \quad (10)$$

because,
$f(0) = e^0 = 1$
$f'(0) = e^0 = 1$
$f''(0) = e^0 = 1$, etc.

Young's First Order of Approximation.

Young's First Order of Approximation is specifically based on the explicit requirements for calculating checkpointing intervals. It takes into considerations factors that affect the checkpointing of applications during their execution to calculate the checkpoint intervals, which he calls the "optimal checkpointing intervals". For this reason, Young's First Order Approximation was adopted for this research. Young uses Maclaurin expansion to derive his First Order Approximation formula. Consider the Figure 4.5 below which shows the execution of an application starting from time t=0 and fails at a time t_i

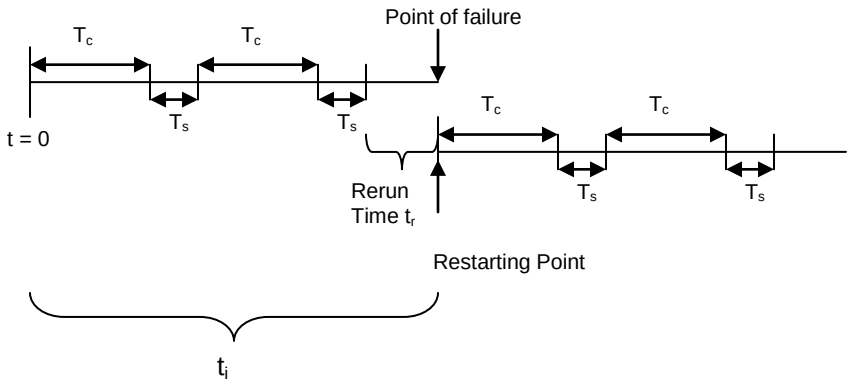

Figure 4.5: First Order Approximation **[43]**

The execution time of the job may be considered as a succession of the intervals T_c, the time interval between checkpoints, and T_s the time to save information at a checkpoint taken alternately until a failure occurs. Such failure may occur during either an interval T_c or T_s. In either case, execution of the program is resumed from the point in the program at which the previous checkpoint was successfully taken. The rerun or recovery time t_r incurred due to the occurrence of such failure is then the time from the end of the previous interval T_s to the point of failure, as shown on the time line in Figure 4.5 **[43].** Accordingly to Young, when the length of the interval t_i between failures lies between: $n(T_c + T_s)$ and $(n + 1)(T_c + T_s)$, n = 0, 1, . . . , then t_i is composed of n intervals of length $(T_c + T_s)$, plus the rerun time t_r. That is,

$$t_i = t_r + n\,(T_c + T_s) \qquad (11)$$

The extra time spent (t_l) due to the occurrence of the failure consists of the time n T_s taken to do the checkpointing prior to the occurrence of the failure, plus t_r. That is,

$$t_l = n\,T_s + t_r \qquad (12)$$

From the previous expression this may be written as

$$t_l = t_i - nT_c. \qquad (13)$$

In the Grid environment, the occurrence of failures tends to be unpredictable. So, sticking to Young's mathematical assumption, we take the occurrence of failures as a Poisson process, with failure rate λ. Based on this, the mean time T_f between failures is $T_f = 1/\lambda$, and the density function $P(x)$ for the time interval of length x between failures is given by $P(x) = \lambda e^{-\lambda x}$ which means that the probability that the interval between failures is of length t_i is given by $\lambda e^{-\lambda t_i} \Delta t$, where $t < t_i < t+\Delta t$ **[43]**.

But the probability that the interval between failures is of length t_i is precisely the probability that the lost time is of duration t_l. Therefore, denoting by T_l the total time lost due to reruns caused by failures and to the time required for the process of checkpointing prior to the occurrence of failures, we have

$$T_l = \sum_{n=0}^{\infty} \int_{n(T_c+T_s)}^{(n+1)(T_c+T_s)} [t - nT_c] (\lambda e^{-\lambda t}) dt$$

$$= \lambda \int_{0}^{\infty} t e^{-\lambda t} dt - \lambda T_c \sum_{n=1}^{\infty} n \int_{n(T_c+T_s)}^{(n+1)(T_c+T_s)} e^{-\lambda t} dt \qquad (14)$$

Integrating, we obtain

$$T_l = 1/\lambda + T_c \sum_{n=1}^{\infty} n [exp(- \lambda (n+1) (T_c+T_s)) - \exp (- \lambda n (T_c+T_s))] \qquad (15)$$

Factoring out the series we obtain

$$T_l = 1/\lambda - T_c (1 - exp(- \lambda (T_c+T_s))) \exp (- \lambda (T_c+T_s)) \sum_{n=1}^{\infty} n exp(- \lambda (n-1) (T_c+T_s)) \qquad (16)$$

The series is in the form of the derivative with respect to r of the geometric series with common ratio.

$$r = \exp(-\lambda\ (T_c+T_s))\qquad (17)$$

Hence its sum is

$$1 / [\ 1 -\ \exp(-\lambda\ (T_c+T_s))\ ]^2\qquad (18)$$

Therefore,

$$T_l = 1/\lambda - T_c\ (1 - exp(-\lambda\ (T_c+T_s)))\exp(-\lambda\ (T_c+T_s)) / [\ 1 -\ \exp(-\lambda\ (T_c+T_s))\ ]^2\qquad (19)$$

which simplifies to

$$T_l = 1/\lambda + T_c\ /\ [1 - exp(\lambda\ (T_c+T_s))\ ]\qquad (20)$$

To find the value of T_c which minimizes T_l , we differentiate T_1 with respect to T_c , equate the result to zero, and solve for T_c

$$\frac{dT_l}{dT_c} = \frac{1 - exp(\lambda\ (T_c+T_s)) - T_c\ (-exp(\lambda\ (T_c+T_s)))}{[1 - exp(\lambda\ (T_c+T_s))\ ]^2} = 0\qquad (21)$$

This means that:

$$e^{\lambda Tc}\, e^{\lambda Ts}\, (1-\lambda T_c) - 1 = 0\qquad (22)$$

Using the Maclaurin expansion of $e^{\lambda Tc}$ as far as the second degree term because T_c is small, we get:

$$e^{\lambda Tc} = 1 + \lambda T_c + \lambda^2 T_c^{\,2}/2! = 1 + \lambda T_c + \lambda^2 T_c^{\,2}/2\qquad (23)$$

Replacing in equation above, we get

$$(1 + \lambda T_c + \lambda^2 T_c^2/2)\, e^{\lambda Ts}\, (1-\lambda T_c) - 1 = 0 \quad (24)$$

Simplifying,

$$(1 + \lambda T_c + \lambda^2 T_c^2/2)\, (1-\lambda T_c)\, e^{\lambda Ts} = 1 \quad (25)$$

$$(1-\lambda T_c + \lambda T_c - \lambda^2 T_c^2 + \lambda^2 T_c^2/2 - \lambda^3 T_c^3/2)\, e^{\lambda Ts} = 1 \quad (26)$$

Ignoring degrees above the second degree, we get:

$$(1-\lambda T_c + \lambda T_c - \lambda^2 T_c^2 + \lambda^2 T_c^2/2)\, e^{\lambda Ts} = 1 \quad (27)$$

Further simplifying,

$$(1 - \lambda^2 T_c^2/2)\, e^{\lambda Ts} = 1. \quad (28)$$

$$1 - \lambda^2 T_c^2/2 = 1/ e^{\lambda Ts} \quad (29)$$

$$1 - \lambda^2 T_c^2/2 = e^{-\lambda Ts} \quad (30)$$

$$1 - e^{-\lambda Ts} = \lambda^2 T_c^2 / 2 \quad (31)$$

Therefore, the expression becomes:

$$1/2\, \lambda^2\, T_c^2 = 1 - e^{-\lambda Ts} \quad (32)$$

Since $T_f = 1/\lambda$, and in practice $T_s << T_f$, we may use the second order approximation to $e^{-\lambda Ts}$ to obtain

$$e^{-\lambda Ts} = 1 - \lambda T_s + \lambda^2 T_s^2 /2! = 1 - \lambda T_s + \lambda^2 T_s^2/2 \quad (33)$$

Therefore, replacing in equation 32, we get:

$$1/2\ \lambda^2\ T_c^2 = 1 - (1 - \lambda T_s + \lambda^2 T_s^2/2) \quad (34)$$

Simplifying, we get

$$T_c^2 = 2\ T_s/\lambda\ - T_s^2 \quad (35)$$

Replacing l/λ by T_f , we get

$$T_c^2 = 2T_sT_f - T_s^2. \quad (36)$$

Neglecting the term T_s^2 as being of second order with respect to $2T_sT_f$, we obtain

$$T_c = \sqrt{2T_sT_f} \quad (37)$$

Where

T_s is the time required to save information at a checkpoint (Ts).

T_f is the mean time between failures (T_f)

This is a simple result, and easy to apply in practice **[43]**.

As mentioned earlier, the Grid environment refers to the Internet-connected computing environment in which computing and data resources are geographically distributed in different administrative domains with different policies for security and resource uses. The computing resources are heterogeneous, ranging from single PCs and workstations, cluster of workstations, to supercomputers. With Grid technologies large-scale applications can be constructed over the Grid environment. However, the execution of applications on the Grid environment poses significant challenges due to the diverse

failures and error conditions encountered during execution. Failures or error conditions occur mainly because of the unreliable nature of the Grid environment. Examples of failures and error conditions include hardware failures, software errors and many other sources of failures (e.g. network congestion, excessive CPU load, etc). Certain type of errors can be detected before execution of an application mainly if these errors are at an application level. However, because of the nature of the Grid, it is difficult to have total control on the resources where applications are being executed. Therefore it can be difficult to detect errors associated to these resources before the execution of the application. This may lead to unpredictable failures during the execution of the application. That is we cannot say for sure if the application will complete successfully or may fail at certain points. It is also difficult to predict how many times the application may fail and when a failure is likely to occur **[11].**

4.3 The Improved Checkpoint Mechanism

Taking checkpoints at intervals defined by the First Order Approximation still involves synchronisation of messages and capturing in-transit messages. On the other hand, taking checkpoint at natural synchronisation points only may not be very effective because there are no patterns in their occurrences. There can be situations where a set of natural synchronisation points occur in quick successions. It is not efficient to take checkpoint at each of these points because it will affect the performance of the application. There can also be situations where we have long periods between two successive natural synchronisation points and not taking a checkpointing for a long period reduces the reliability of the application.

A better solution would be to use a combination of both the natural synchronization points and the First Order Approximation before making a checkpointing decision. This mechanism is named the “Improved Checkpoint Mechanism”. Using this technique, the most appropriate places to take checkpoints can be selected. The solution takes checkpoints at natural synchronization points which are closest to the Young’s checkpoint intervals. The Figure 4.6 below explains how the checkpointing intervals are chosen. The vertical lines represent the Young’s checkpointing intervals and the natural synchronization points. The bracket represents the critical region; a region within which a

checkpoint may be taken. First, the checkpoint interval needs to be calculated using the First Order Approximation formula. Then, the natural synchronization points (Barriers, iteration and collective information) in the application need to be determined so that checkpoint calls can be inserted where appropriate.

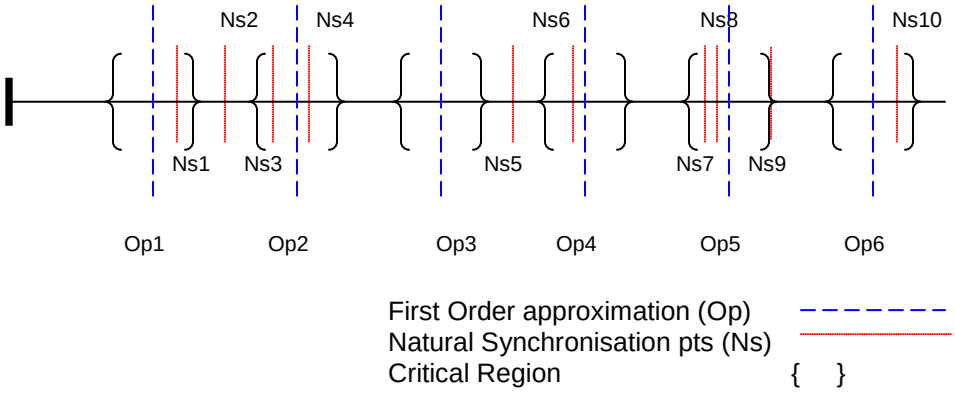

Figure 4.6: A Checkpointing Mechanism

The decision to select a checkpoint is based on the Young's checkpoint interval, the natural synchronisation points and the critical region. Whenever an application receives a checkpointing signal, we may need to take a checkpoint. The checkpointing process is triggered by signals sent to the coordinator process whenever synchronization points are encountered. Once the coordinated process receives a signal, it checks whether this signal is within the critical region. If not, no checkpointing is performed. However, if the signal occurs within the critical region, we will need to take a checkpoint. Within that region, there may be more than one natural synchronization points and the one closest to Young's checkpointing interval is the best choice. For our purpose, the checkpoint image at the first natural synchronization point encountered is saved. This is because we cannot predict if we will get a better solution further along the execution line within that critical region. If no natural synchronization points are met within the critical region, we will have to force a checkpointing at the end of the critical region. In such cases, the

checkpointing mechanism will perform synchronization to ensure there are no lost or orphan messages. The coordinated process will make sure that the checkpointing images together with the in-transit messages are saved. Once the checkpoint is taken, all the processes will resume their normal execution. In case of a failure, the coordination process will initiate the rollback mechanism. If the checkpoint to be restored was taken at a natural synchronization point, the rollback mechanism will load each process image from the checkpointing file and the execution process is resumed. We do not have to worry about the in-transit messages. However, if the checkpoint to be restored was not from a natural synchronization point, then we will need to restore the in-transit messages as well to ensure consistency.

For example, consider the model solution shown in Figure 4.7 below.

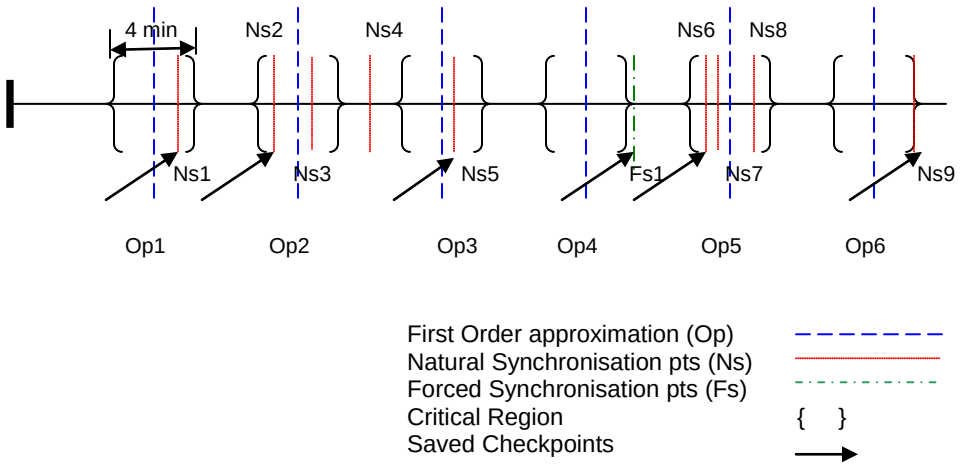

Figure 4.7: The Checkpointing Solution

Assume that, through the First Order Approximation, the calculated checkpoint interval was 8 minutes and a critical region of 4 minutes around the checkpoint interval was defined.

Based on the proposed methodology, the first checkpoint that is stored is Ns1 (9.5 min). As execution continues, we reach Ns2 (14.5 min) where a decision has to be taken on whether or not to take the checkpoint. Ns3 (17 min) is a better solution because it is

nearer to the Young's checkpoint interval Op2. However, since we cannot forecast what will happen, it is best to take a checkpoint at Ns2. As we move towards Op3, we get the natural synchronisation point Ns4. Because it is outside the critical region it is not considered. As we move on, we enter another critical region and store the checkpoint at Ns5 (25 min). Unfortunately, within the fourth critical region there are no natural synchronisation points. In that case, we need to force a checkpoint (Fs1 - 32 min) as soon as we leave that critical region. The program continues execution and enters another critical region which contains the natural synchronisation points Ns6 (38.5 min), Ns7 (39 min) and Ns8 (41.5 min). However, a checkpoint is taken at the first synchronisation point, Ns6. As the program continues, another checkpoint is taken at Ns9 (50 min).

From the First Order Approximation, we deduced that best option for checkpoint intervals should be 8 minutes. If checkpoints are taken at selected points based on the proposed solution, the average time between checkpoints is 8.3 minutes. However, here, less time was needed to save the checkpoints. So, the overall execution time for the application in a failure free environment will be better. It also implies that in case of failure the application will be restarted quicker as we will not have to worry about restoring the communication messages. Therefore, the proposed solution provides a better checkpointing solution with improved performance.

4.4 The Improved Checkpoint Model

The Figure 4.8 below gives a generalised description on how a checkpoint at a natural synchronisation point will be taken.

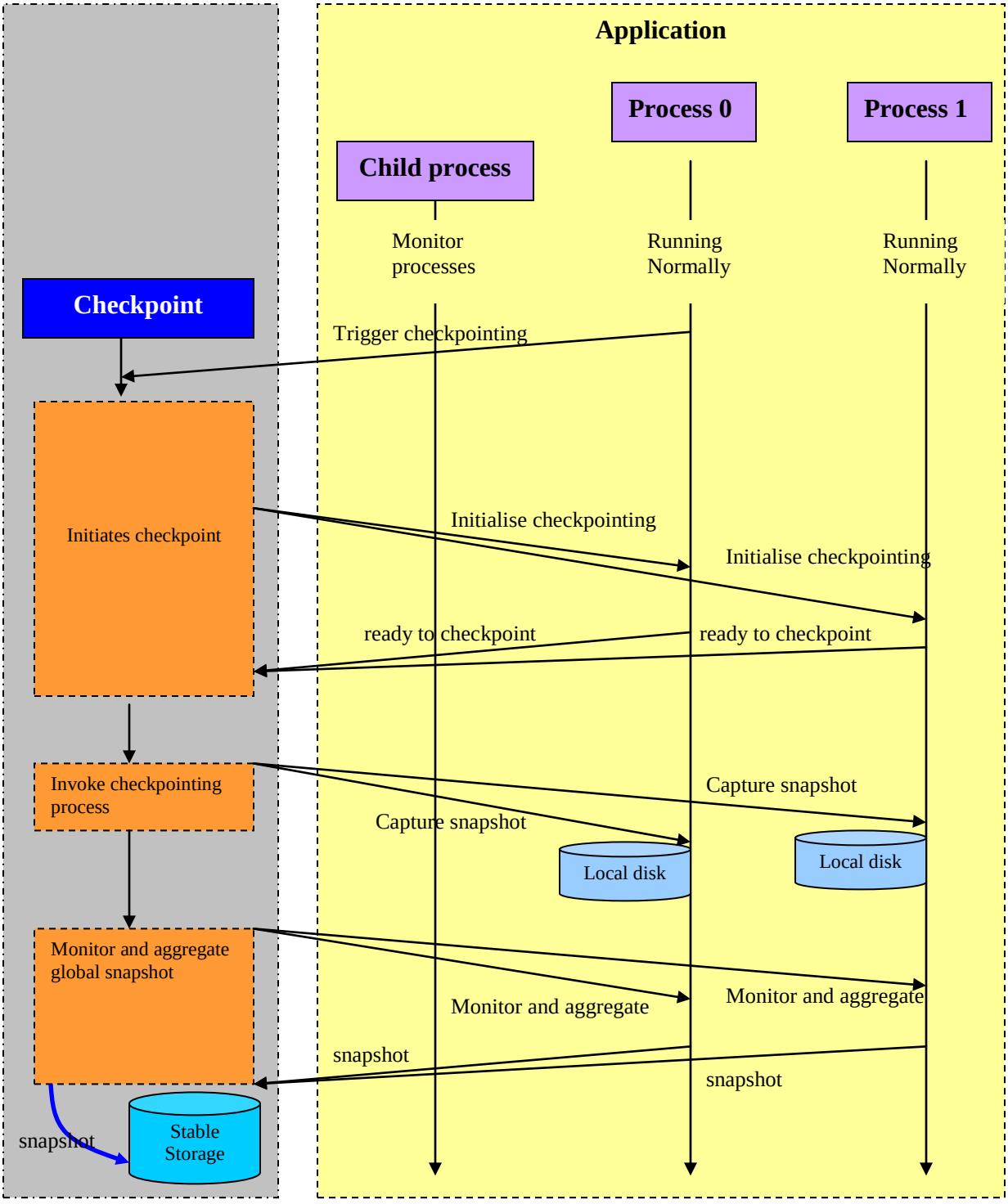

Figure 4.8: Checkpointing process at Natural Synchronisation Points

The Figure 4.9 below shows how a forced checkpoint can be taken.

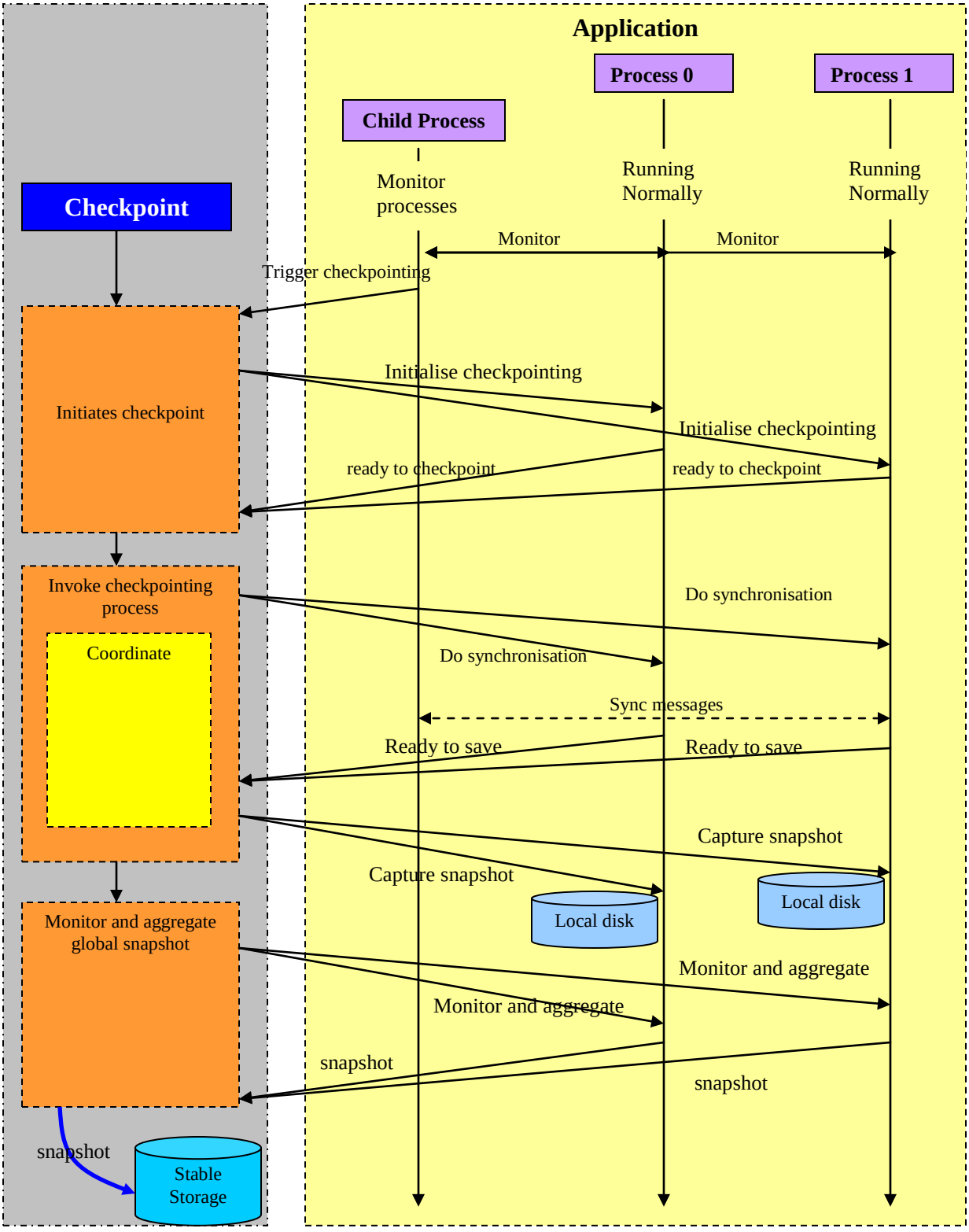

Figure 4.9: Taking a forced checkpoint

The Figure 4.10 below shows how the restart process will happen.

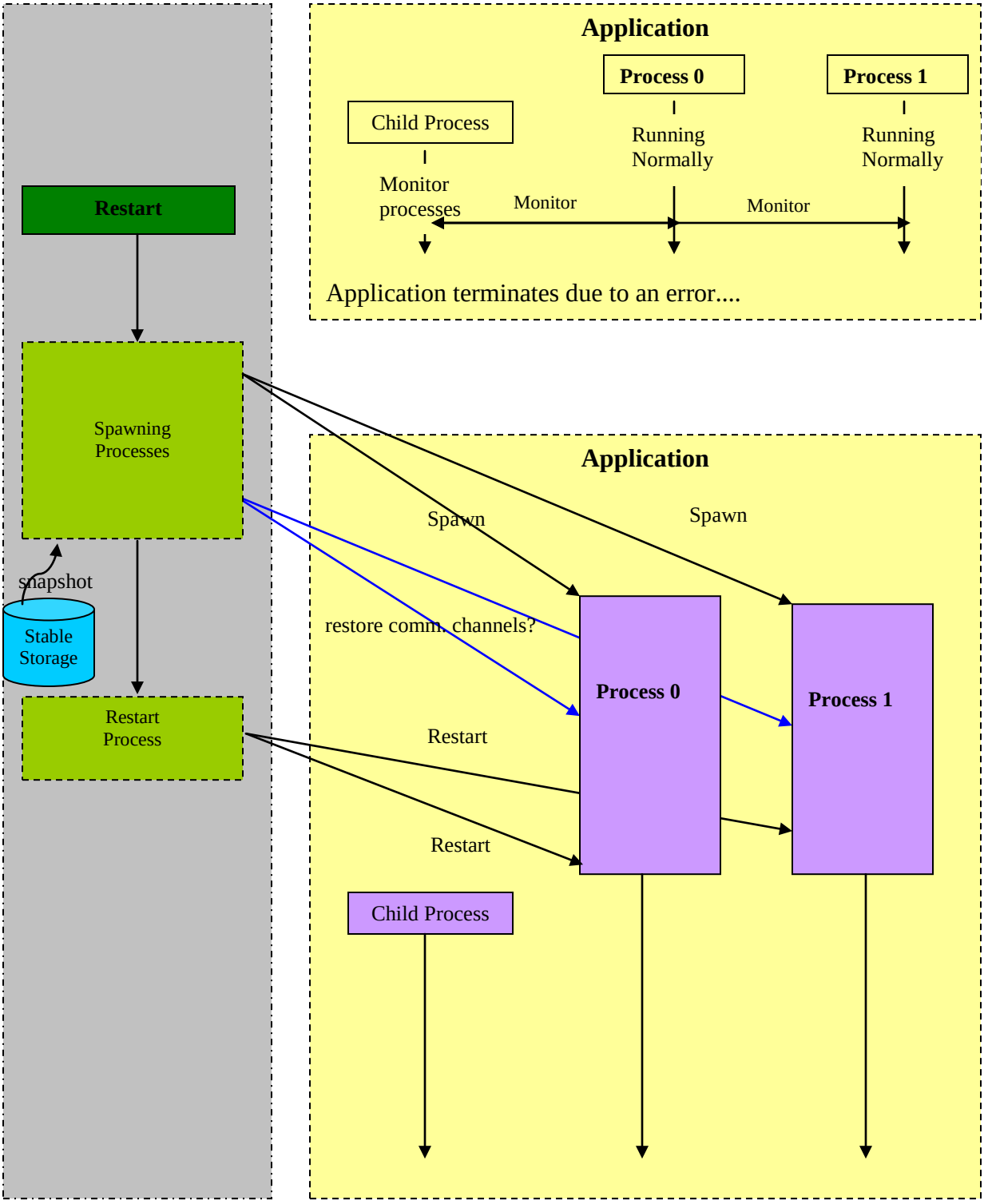

Figure 4.10: The Restart Process

4.5 The Improved Checkpoint Algorithms

To checkpoint an application at natural synchronization points, the first step is to find out the Young's Checkpoint Time (OCT) which is determined by the First Order Approximation. While the application is running, we might get a few natural synchronization points. At a given time, determined by TIME2-TIME1, if we get a natural synchronization point, then we need to decide if we are going to take a checkpoint. A checkpoint will only be taken if we are in a critical region and no checkpoint (no_of_checkpoints =0) has yet been taken in that region. If this is the case, the checkpointing mechanism is triggered to initiate the checkpointing process. Because we are checkpointing at natural synchronizations points, we don't need to coordinate the communication channels to manage in-transit messages. So, the next step is to save the snapshot returned. Finally, the no_of_checkpoints is incremented so that if we get another natural synchronization point in the same critical region, we do not take another checkpoint.

Algorithm 1: Checkpointing:: NSP Algorithm

```
1: OCT: Calculated integer value using FOA
2: Initial Execution Region: 0
3: TIME1 : start time
4: TIME2 : current execution time.
5: Current Region: NULL
6: Critical Region: No
7: No_of_checkpoint: 0
8: Current Region ← TIME2 - TIME1
9: Critical region ← Region(Current Region)
10: if Critical Region = yes then
11:     if no_of_checkpoint = 0 then
12:         Initiate Checkpoint Process
13:         (snapshot) ← CHECKPOINT()
14:         no_of_checkpoint = no_of_checkpoint + 1.
```

15: **else**
16: do nothing.
17: **end if**
18: **end if**

In case of a forced checkpoint, the algorithm is quite similar to the above algorithm expect that in the case, we need to ensure synchronization of in-transit messages. So, once we reach the end of a critical region (End_Current_Critical_Region), we need to check whether there has been a checkpoint in that critical region. If no checkpoint was taken, we need to force a checkpoint. When the checkpointing mechanism is triggered, the checkpointing process is started. In this case however, we need to manage in-transit messages. Once done, the snapshot returned by the CHECKPOINT API is saved.

Algorithm 2*:* Checkpointing:: Forced Checkpoint Algorithm

1: OCT: Calculated integer value using FOA
2: End_Current_Critical_Region: no.
3: TIME1 : start time
4: TIME2 : current execution time.
5: Current Region: NULL
6: Critical Region: No
7: No_of_checkpoint: 0
8: Current Region ← TIME2 - TIME1
9: Critical region ← Region(Current Region)
10: **if** Critical Region = no **then**
11: **if** End_Current_Critical_Region = yes **then**
12: **if** no_of_checkpoint = 0 **then**
13: Initiate checkpointing process
14: coordinate checkpoint
15: (snapshot) ← CHECKPOINT( )
16: no_of_checkpoint = no_of_checkpoint + 1.

```
17:          else
18:              do nothing.
19:          end if
20:      end if
21: end if
```

To restart an application from a checkpoint file, a RESTART API should be executed. The checkpoint/restart mechanism will check if it has received a request to restart the application. Then the RESTART function will spawn the processes. In case the checkpoint was taken at a natural synchronization point, the return the TYPE will be "NSP". If so, the application will start executing straight after. If the TYPE is "FORCED", then the checkpoint/restart mechanism will need to restore and coordinate the communication channels before the application can resume.

Algorithm 3: Restarting:: Restarting an application after a failure

```
1: STATE:NULL
2: (TYPE) ← RESTART ()
3:     If TYPE=NSP then
4:      Restore process state
5:      Start execution
6:    else
7:         Restore process state
8:         Restore in-transit messages
9:        start execution
10: end if
```

4.6 Applicability and Suitability of the Proposed Algorithm

In a Grid environment, checkpointing is a technique that helps tolerate the errors leading to losing the effect of work of long-running applications. The main property which should be induced by checkpointing techniques in such systems is in preserving system consistency in case of failure. Nowadays, Grid computing is used in a wide range of fields, from bioinformatics to economics. A range of applications can be executed on the Grid including Barnes–Hut simulation, Monte Carlo simulation, Brute-Force cryptographic techniques and Finite-State machine simulation among others **[61]**. Furthermore, many research institutions are using Grid computing to address complex computational challenges. The University of Westminster, for example, has been testing its GEMLCA Grid environment using the "Urban Car Traffic Simulation". Therefore, the business companies or research institutions will need a certain degree of fault tolerance while executing the application on the Grid environment.

The proposed checkpointing solution is suitable for MPI applications executed in a Grid environment. Therefore, business companies and research institutions who are performing extensive computations/simulations on the Grid environment using MPI based parallel applications can easily adopted the proposed checkpointing solution to make their application fault tolerant. All they need to do is to link their application to the proposed solution by importing the libraries written and do some minor modifications to their application to support the checkpointing mechanism. This will ensure that their application execute reliably in the Grid environment while at the same time incurring minimal overhead. This is very beneficial as it will provide a performance benefit solution, which is the main aim of the Grid.

4.7 Chapter Summary and Conclusions

The chapter starts by describing the proposed solution and the different components that are part of the solution such as the First Order Approximation and the Natural synchronisation points. It then gives design models of the proposed solution explaining how the checkpointing process at a natural synchronisation point and at a non natural

synchronisation point will work. It also gives a design model of the restarting process showing how application will be restored from a global snapshot. Finally, it describes the checkpointing and restart algorithms and gives an overview of the applicability and suitability of the proposed algorithms.

The proposed checkpointing mechanism is an innovative way to checkpoint parallel applications running on a Grid environment. The solution takes into consideration the necessity to execute long running applications reliably keeping in mind the need to reduce checkpointing overhead to improve performance. The solution exploiting different areas to achieve its goal mainly the natural synchronizations points that exist in parallel/distributed applications and the regular checkpointing intervals determined using the First Order Approximation. A critical region has been defined in order to ensure that the checkpoints taken during the execution of an application does not deviate too much from the Young's checkpointing interval. Therefore the proposed solution provides a better and more efficient way to perform the checkpoint/restart process. The next step is to implement the mechanism to provide an efficient and reliable fault tolerant solution to applications executed on Grids.

Chapter 5

5 Implementing the Improved Checkpoint Mechanism

5.1 Introduction

In this chapter, the steps taken to implement the proposed checkpointing model are explained. The first section explains openMPI and BLCR which was adopted to implement the proposed solution. The second section further elaborates on the design models explained in chapter 4. The last section explains the implementation in brief, showing the pseudo code for the implemented solution.

5.2 The OpenMPI Architecture

Having proposed a checkpointing solution, the next step was to design and implement the solution. OpenMPI with Berkeley Lab Checkpoint/Restart (BLCR) **[42]** was a good starting point to implement the proposed solution rather that starting from scratch. In this section OpenMPI is described **[40] [41].**

5.2.1 The OpenMPI Model

In chapter 3, the OpenMPI architecture and the checkpoint and restart service framework was briefly explained. In this section, the functionality of the model that makes it a viable solution for the proposed checkpointing solution was explored.

Figure 5.1 below shows the fault tolerant solution provided by openMPI to ensure MPI applications are executed reliably in a distributed environment. The components involved are: The Snapshot Coordinator (SnapC), Remote File Management (FileM),

Checkpoint/Restart Coordinate Protocol (CRCP), Interlayer Notification Callback (INC), Checkpoint/Restart Service (CRS) **[41].**

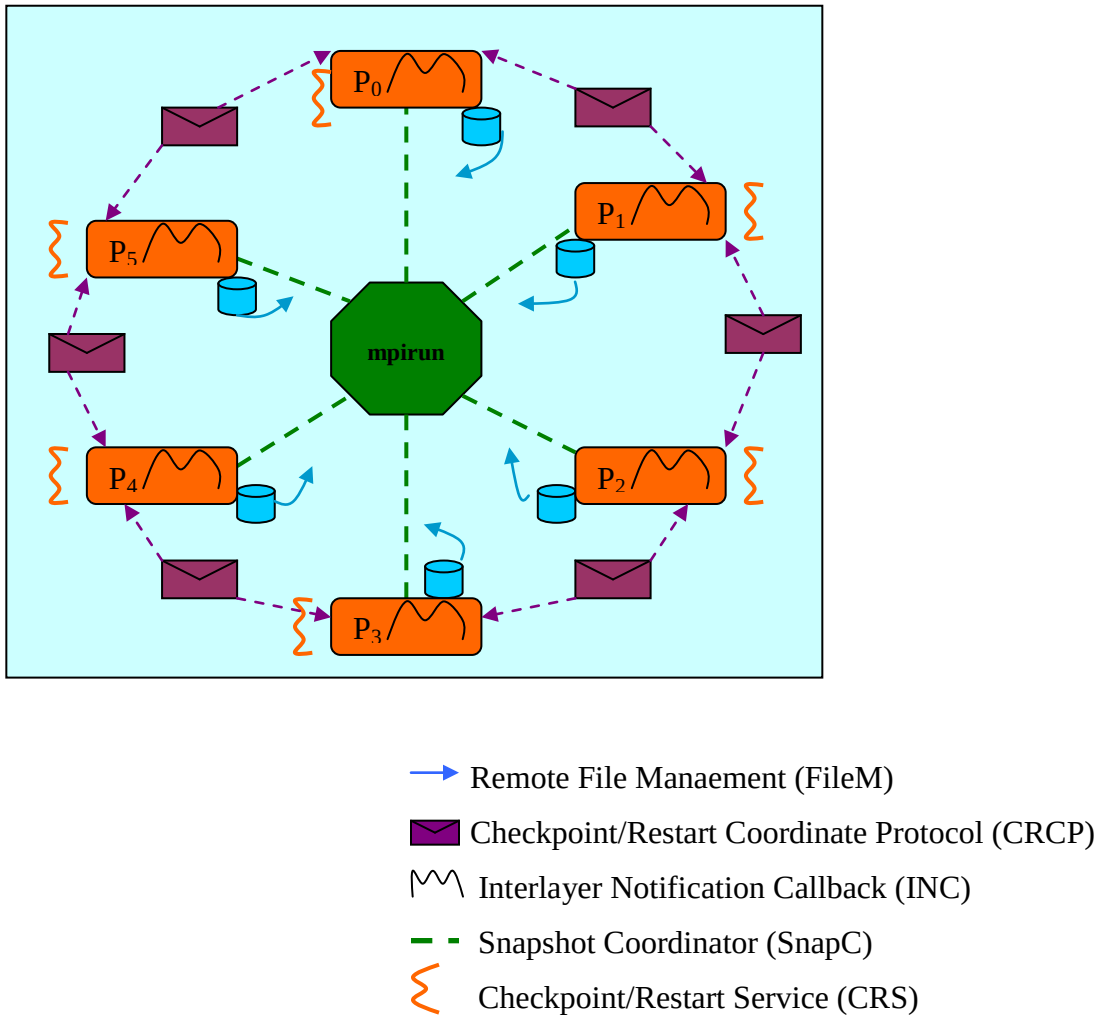

Figure 5.1: A process fault tolerance infrastructure for Open MPI **[40]**

The Snapshot Coordinator

Upon receiving a checkpoint/restart request, the Snapshot Coordinator will initiate the local checkpoint operation for each process. The Snapshot Coordinator will also monitor the checkpointing process to ensure that all the local checkpoints have successfully completed. Once the local checkpoints are taken snapshot coordinator aggregates the local checkpoints into a global checkpoint file and save it to the stable storage **[40].**

Interlayer Notification Callback (INC)

Single process checkpoint/restart services usually exclude the state of open files and communication channels from the checkpoint file of the process. As a result, the different layers within Open MPI need to receive notification surrounding all checkpoint and restart requests. The main layers are: OPAL, ORTE and OMPI as explained in chapter 3. Each of these layers requires notification during the checkpointing and restart process. So, they register an INC providing each layer ample opportunity to take necessary action around a checkpoint/restart request **[41].**

Checkpoint/Restart Coordination Protocol (CRCP)

For a parallel/distributed application, a checkpoint file consists of the state of the processes and all associated communication channels. Single process checkpoint/restart services do not preserve the state of the communication channels. So, the CRCP protocols ensure a consistent distributed execution during process failure and recovery by managing the communication messages **[41].**

Checkpoint/Restart Service (CRS)

This is a framework for single process checkpoint/restart services. Services must capture a snapshot of a single process on the local system, save the snapshot to the stable storage, and be able to restart the process from that snapshot if a failure happens.

The CRS framework, also known as the OPAL CRS MCA framework, provides a simple API for the Open MPI layers to interact with a checkpoint/restart service. Berkeley Lab Checkpoint/Restart (BLCR) is the checkpoint/restart mechanism that OpenMPI uses by default. The two main functions of the API are CHECKPOINT() and RESTART(). These functions enable Open MPI to request a checkpoint internally as well as still retaining support for user requested checkpoints via command line tools **[42].**

```
int CHECKPOINT( pid_t pid, snapshot_handle_t *snapshot, int *state);
int RESTART( snapshot_handle_t *snapshot, bool spawn_child, pid_t *child_pid);
```

The CHECKPOINT function initiates the checkpoint of a single process, identified by its process ID (PID), by calling the checkpoint/restart mechanism's checkpoint routines. This function returns a snapshot handle representing the snapshot reference. This function also returns the state of the system following the checkpoint that is used by the interlayer coordination callbacks (INC). The state is expected to be one either CONTINUE or RESTART. If after CONTINUE, it implies that the application will continue its execution after the checkpoint has been taken. If the state is RESTART, the RESTART function will initiate the restart of a single process from a snapshot reference by interacting with the checkpoint/restart mechanism's restart functionality **[40][41].**

To trigger a checkpoint or restart a checkpointed MPI application, Open MPI uses: ***ompi checkpoint and ompi restart.*** To send a checkpoint request to mpirun, the user specifies the PID of the application to the ompi checkpoint command:

```
shell$ ompi_checkpoint [OPTIONS] mpirun_pid
```

When this command completes, the user is presented with a string name referencing the global snapshot that can be used to restart the parallel application. To restart the parallel application, the user specifies the global snapshot name to the ompi restart command, as seen below.

```
shell$ ompi_restart [OPTIONS] global_snapshot_reference
```

Berkeley Lab Checkpoint/Restart (BLCR)

As mentioned earlier in this sub section, OpenMPI uses BLCR as the default checkpoint/restart mechanism. BLCR provides a checkpoint/restart solution on Linux systems. It can be used either with processes on a single computer, or on parallel jobs such as MPI applications which may be running across multiple machines on a cluster. The Checkpoint/Restart mechanism allows you to save one or more processes to a file and later restart them from that file. To checkpoint an application, it needs to be safely stopped at any point in its execution, so that a checkpoint can be taken.

To initiate a checkpoint within an application, BLCR sends it a signal. User applications that need to be checkpointed must be loaded with the BLCR checkpoint library, which registers a signal handler for the checkpoint signal **[42].**

Remote File Management (FileM)

This framework provides the ability to transfer local process snapshots to and from stable storage device(s) as required by the checkpoint/restart mechanism **[41].**

5.3 Designing the Improved Checkpoint Mechanism

In order to implement the proposed solution, a few functions had to be written that would then be integrated at the application level. Therefore, some modifications had to be made to the MPI application to support the proposed solution.

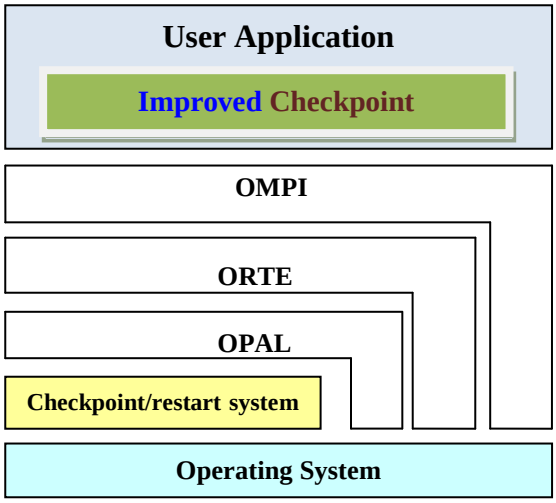

Figure 5.2: The improved OPENMPI Architecture

The integration of the proposed solution is simple. All the users need to do is to include the implemented API in their MPI application. This can be achieved either by adding the functions in the c file or storing them in a library and calling the library file. The only changes to the MPI application would be to include the ***mychkpt( )*** function at natural synchronisation points. The functions are explained further below.

The Improved Checkpoint Model

The Figure 5.3 below describes how a checkpoint at a natural synchronisation point is taken.

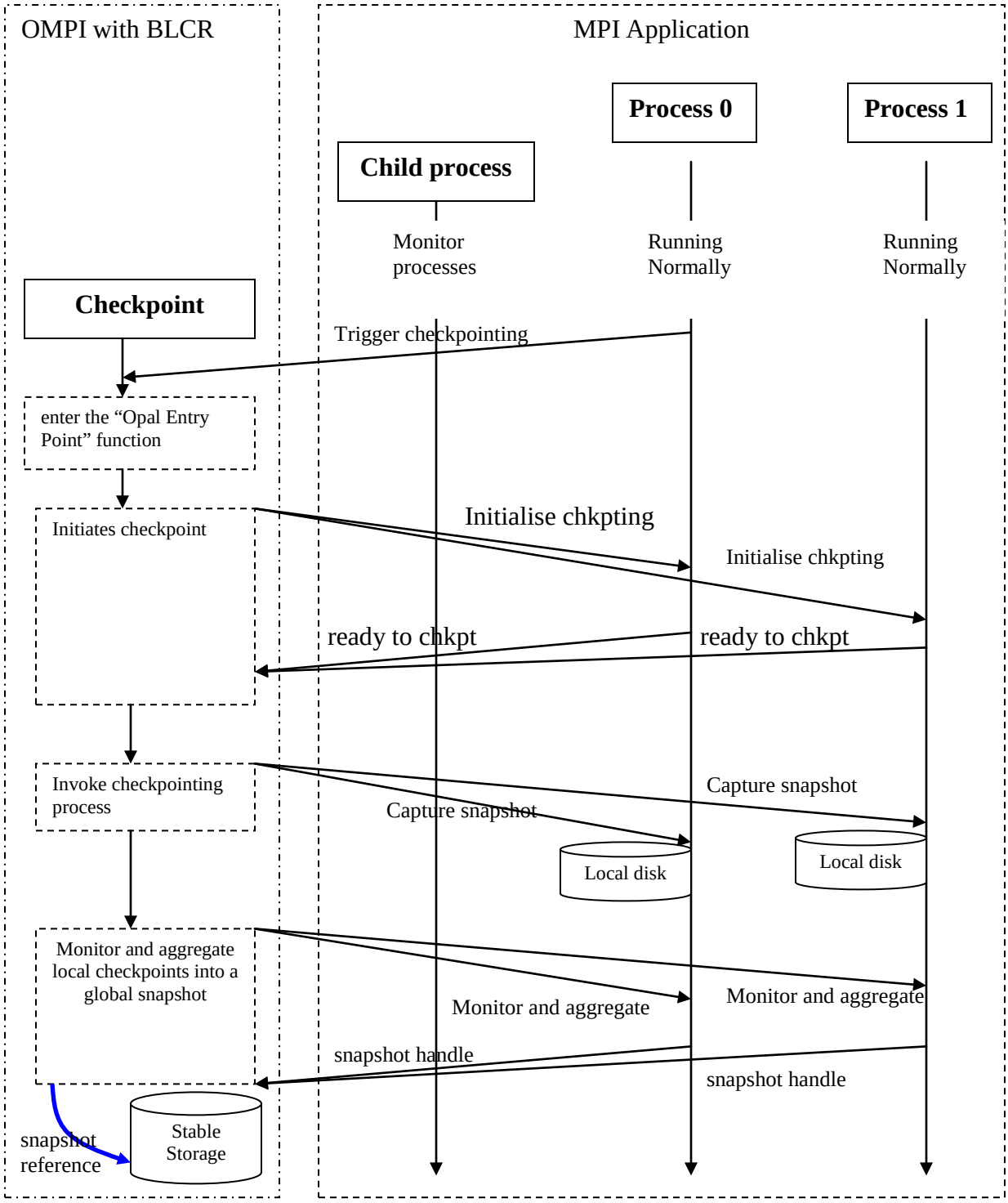

Figure 5.3: checkpointing process at Natural Synchronisation Points

The figure below describes how a forced checkpoint is taken.

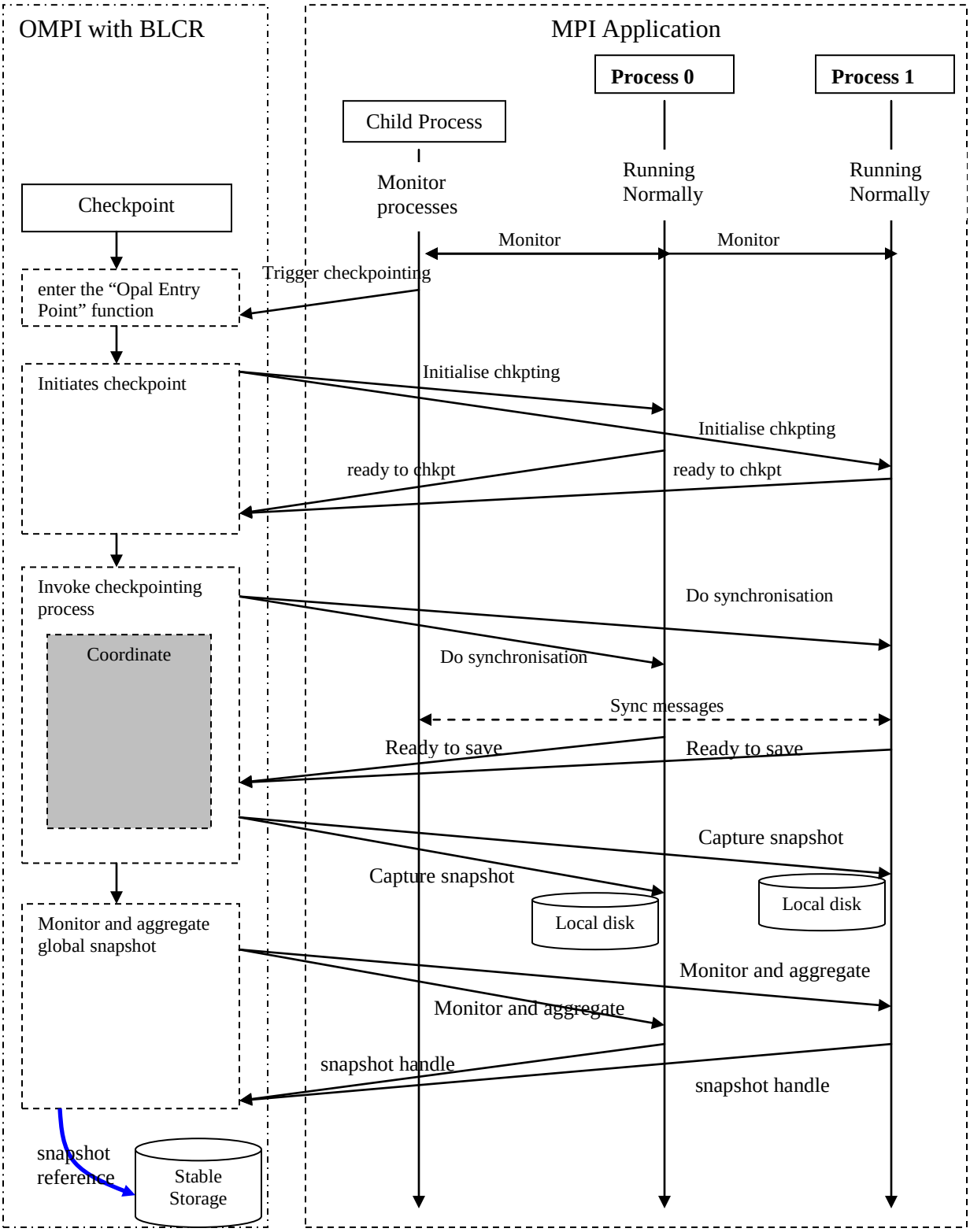

Figure 5.4: Taking a forced checkpoint

The figure below describes how the restart process happens.

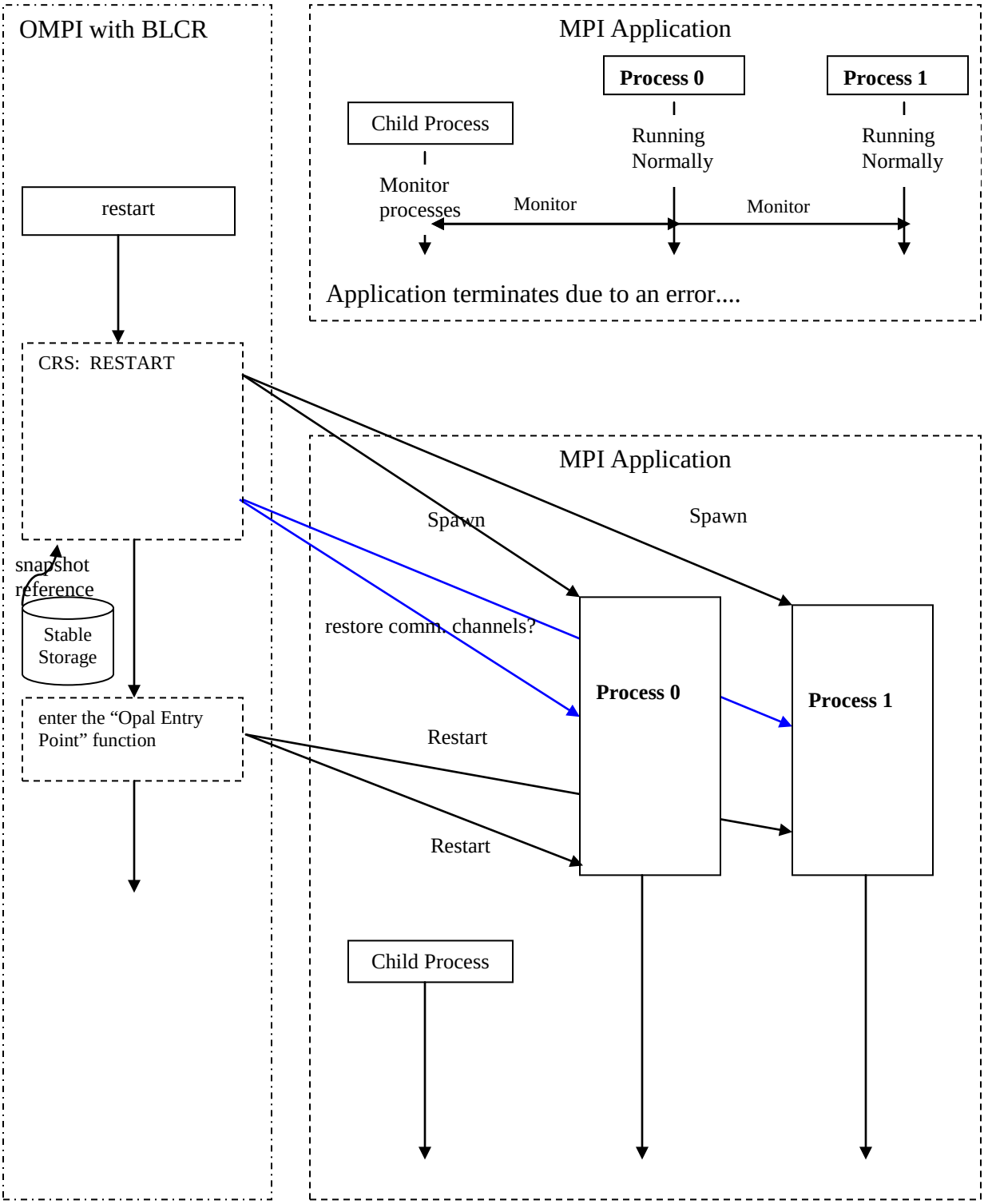

Figure 5.5: The Restart Process

5.4 Implementing the Improved Checkpoint Mechanism

Having developed the model, the next step was to write the APIs that would allow us achieve our desired goal.

The createcheckpointthread() function

The *createcheckpointthread()* function was created a child thread which is used to monitor the main application and triggering a forced checkpoint when required by calling the myfirstthread() function.

```
void createcheckpointthread(int rank,int argc, char **argv)
{
Define a pthread;
Assign predefined values to an array of type struct. In this array, we will specify the optimal checkpoint interval.
  if(rank==0) /* i.e. in main */
     {
      Create a pthread and call the myfirstthread() routing passing the array data as
      argument.
     }
}
```

The myfirstthread() function

The myfirstthread*()* function is executed by a child thread created by the main application. It is responsible for triggering the forced checkpoints.

```
void *myfirstthread(void *threadarg)
{
Assign arguments to new variable of the thread_data; /* the optimal checkpoint interval value is passed to this function.

 struct thread_data *my_data;
 calculate criticalendrange  /*if optimal =701s, then criticalendrange = 175s */
 myfirstextrasleeptime = 175 + 1;
```

```
        for(;;)
    {
      Get time passed since the program has started.
       time_previous_checkpoint_taken=time_new_checkpoint_taken; /* if the program
       has just started, both values are 0 */
      previous_checkpoint_count=new_checkpoint_count; /* this gives the number of
      checkpoints taken during a cycle. */

       if(loopcount==0)
       {

                          time_new_checkpoint_taken = mytime;
                           Force another checkpoint.
                  }
                  else
                  {
                      A checkpoint has already occurred in the critical region.

                  }
          }

     loopcount++;
    }

}
```

The mychkpt() function

To trigger a checkpoint at natural synchronization points within the critical region, the mychkpt() function was implemented. The "***Algorithm 1***" explains how it ensures that a checkpoint is taken only at a natural synchronization point within a critical region. The pseudo code below explains the function.

```
void mychkpt( )
{
   Get the optimal checkpoint time.
   Determine in which region is the program presently running. /* the first cycle is
zero, second cycle is one, etc)
Determine the critical range.

                        if(still within the same cycle)
                           {
                             do nothing to counts.
                           }
                          if (move to the next cycle)
                            {
                              /* this is to ensure that we don't take a checkpoint if one is
                                 already taken in the same critical range. That is in  the
                                 region before the optimal checkpoint time is reached.

                                 previous_end_range_count = new_start_range_count;
                                 new_start_range_count=0;
                            }
                            if (moved to more than one cycle)
                            {
                             We don't need to worry when the last checkpoint was taken.
                              previous_end_range_count = 0;
                              new_start_range_count = 0;
                            }
                  if((no checkpoint has occurred yet in the present critical region just
after the previous checkpoint interval))
                       {
                          Take a checkpoint by calling: writefifo(rank);
                       }
```

```
                    else
                    {
                       A checkpoint already taken.
                    }
                              previous_d_value=new_d_value;
                       }
                       else if( we are outside a critical region)
                       {
                        Do not take a checkpoint.
                       }
                       else if (it is in the critical region  before the new optimal
checkpoint interval)
                       {
                         Take a checkpoint by calling: writefifo(rank);

                          else
                            {
                             one checkpoint has already been taken.
                            }

                       }

}
```

5.5 Chapter Summary and Conclusions

This chapter summarises the OpenMPI and BLCR application that was adopted to implement the proposed solution. The first section describes the different components of OpenMPI that are important in the checkpointing process. It also explains the BLCR single process checkpointing mechanism. The second section describes the proposed solution giving some design models on how the checkpoint/restart process works.

The third section describes how the proposed solution was implemented, giving pseudocodes of the APIs that was developed.

The OpenMPI was successfully installed on the University of Westminster cluster and the checkpointing solution was successfully implemented and integrated in the MPI applications to be tested.

Chapter 6

6 Experimental Results and Analysis.

6.1 Introduction

In this chapter, the proposed checkpointing solution was tested on a cluster and the outcomes were analysed. The purpose of this chapter was to analyse the efficiency of the proposed solution as compared to other checkpointing techniques. This was achieved by performing a series of testbed experiments followed by an analysis of the results obtained. Therefore, this chapter is divided into two sections - Experimental Results and Analysis.

The first section describes the methodology used for conducting the testbed experiments and consecutively presents the experimental results themselves. The second section provides a discussion based on an interpretation of the experimental results and concludes over the required criteria by which the proposed solution is expected to improve the checkpointing process in Grids.

6.2 Aim

This chapter provides a set of results obtained over a number of tests carried out. The results were analysed to prove the proposed research contribution is viable. The tests described below prove that this research's contribution improves the performance of checkpointing and restarting MPI applications as compared to existing checkpointing methods.

The tests provide concrete results to support the efficiency of the contribution. Through the testing process, we were also able to determine the best "critical region" to further optimise the proposed solution.

6.3 Applications used for Testing

The following two open source MPI applications were used to test the proposed checkpointing solution:

1. PI
2. Fire Reduce

These two applications were chosen because they are open source. Moreover, it was possible to vary the execution time of these applications to suite the testing requirements.
The main difference between the two applications is the occurrence of the natural synchronisation point. In the PI application, the occurrence of the natural synchronisation point is when the collective operation MPI_REDUCE() is called. MPI_REDUCE() occurs at irregular intervals and therefore it is difficult to know when the natural synchronisation point will occur during execution of the application.
The "Fire Reduce" MPI application has a natural synchronisation point at the bottom of a "for" loop in the main section of the application. This means that every time the program loops, we will get a natural synchronisation point. Tests were carried on these two MPI applications to demonstrate that the proposed checkpointing mechanism works well in any type of MPI applications.

PI

This MPI application calculates the value of PI. All processes contribute to the calculation, with the master averaging the values for PI. This version uses a collective operation to collect results before averaging them.
(Collective operations represent natural consistent global states for checkpointing. Examples of collective communication operations are broadcast, scatter/gather and reduction.)

Fire Reduce

This application is a forest fire simulation in which a forest is modelled as an NxN grid of trees. One tree starts to smoulder, and each iteration nearby trees have some chance of catching fire. The simulation runs until the fire burns out.

The main reason for choosing the “PI” and “Forest Fire” MPI applications was because they are open source and the execution time could be varied to the testing requirements.

The problem with these applications is that they are simple programs which take a few seconds to checkpoint and thus a bit more difficult to show improvements in the checkpoint process.

6.4 The Testbed

The tests were carried on the nodes of the University of Westminster (UoW) cluster. It was performed in an isolated environment where the nodes were solely used for testing my research contribution. Each test was repeated at least ten times to minimise side effects that may arise in case some background processes runs randomly.

The Checkpoint Testbed Architecture

Figure 6.1 shows how an MPI application was submitted on the University of Westminster cluster. The MPI application runs on the cluster and the “Improved Checkpoint Mechanism” triggered checkpoint process based on the criteria discussed from the research contribution.

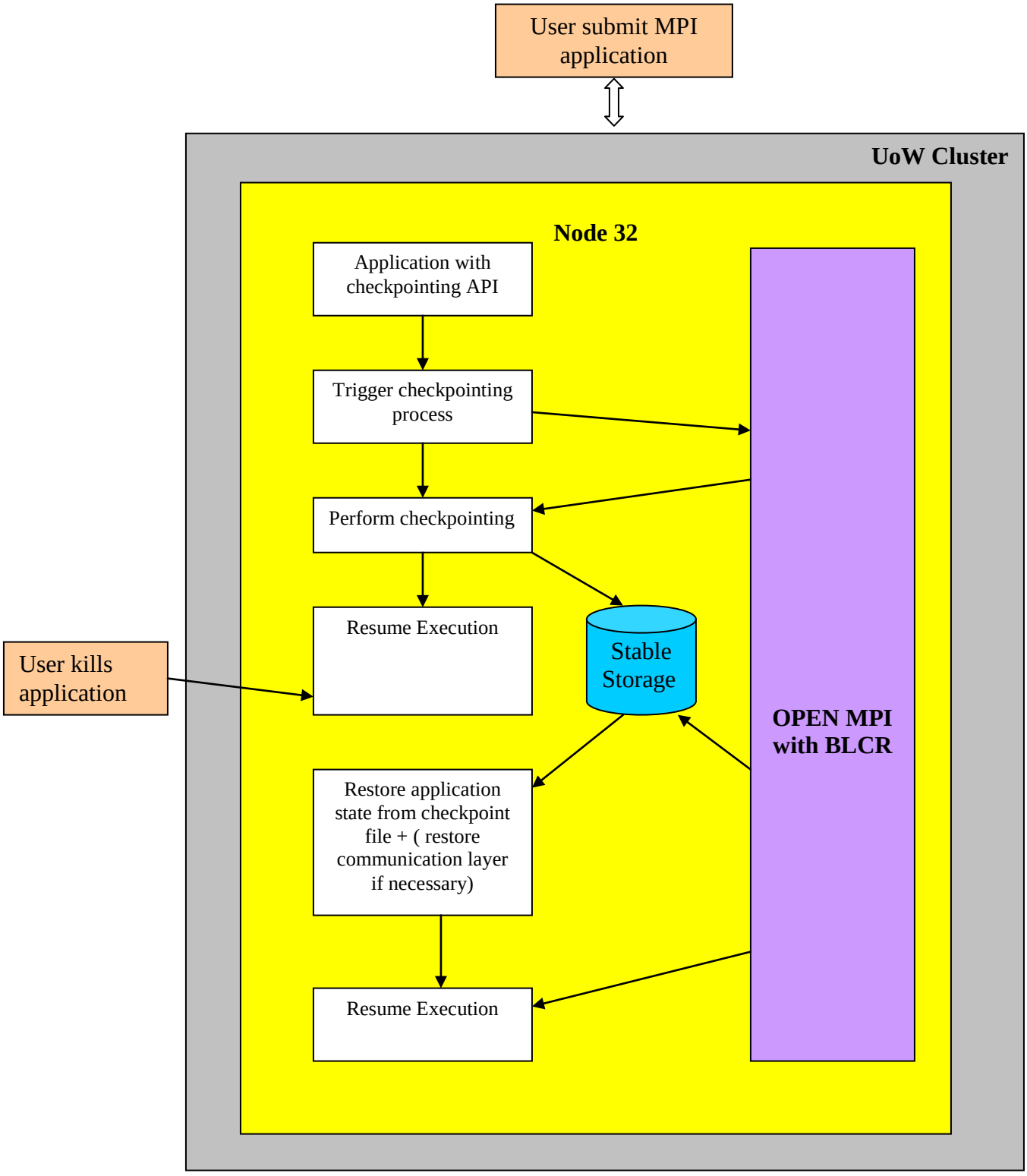

Figure 6.1: Executing and restarting an MPI application on UoW cluster

6.5 Testing Process

Two sets of test for each application were carried out under these conditions:

1. Application executed without any failure occurring.
2. Application execution with failure induced.

To induce failure, the application was manually terminated by killing it. This process caused a failure to the application thus allowing the testing of the proposed checkpoint solution both when the application was executed without failure and with failure. Both set of test were evaluated to determine the efficiency of the proposed solution under the different conditions.

Under each of the above conditions, the following was performed:

1. Running application without performing checkpoints.
2. Running application with checkpoints at ad hoc regular intervals.
3. Running application with checkpoints performed at intervals based on the First Order Approximation.
4. Checkpoint the application at natural synchronisation points (NSP) only.
5. Checkpoint the application using the proposed solution. This was performed three times with varying critical ranges (10%, 25% and 40% of the Young's Checkpointing Interval).

The table 6.1 below summarises the tests that was carried out.

	Execution time without failure	Execution time with failure
No checkpoints	✓	✓
Checkpoint at ad hoc regular interval	✓	✓
Checkpoint at Young's interval	✓	✓
Checkpoint at NSP.	✓	✓
Proposed checkpoint with 10% critical range	✓	✓
Proposed checkpoint with 25% critical range	✓	✓
Proposed checkpoint with 40% critical range	✓	✓

Table 6.1: Categories of tests performed

6.5.1 Data to be gathered during the tests.

For each test performed, the data gathered was used to determine:

1. The average time between checkpoints.
2. The number and type of checkpoints saved. The checkpoint type can either be a checkpoint at a natural synchronisation point or a checkpoint at a non-natural synchronisation point.
3. The recovery time in case of a failure. The tests showed how quickly a failed application recovered under each checkpointing method adopted. The recovery time was obtained by calculating the difference between the execution time without failure and the execution time with failure.
4. Percentage increase in execution time. This showed the increase in execution time when performing checkpointing as opposed to the execution time without taking any checkpoint.
5. The best possible critical range for the proposed solution.
6. Percentage change in time as compared to checkpoints performed at regular intervals.

	Recovery time after failure	% increase in execution time	% change in Time as Compared to Regular Checkpoint	Average Checkpoint Time	% deviation from Young's FOA Checkpoint value	NSP vs forced Checkpoint
No checkpoints	✓	✓	✓	✓	✓	✓
Checkpoint at ad hoc regular interval	✓	✓	✓	✓	✓	✓
Checkpoint at Young's FOA interval	✓	✓	✓	✓	✓	✓
Checkpoint at NSP.	✓	✓	✓	✓	✓	✓
Proposed checkpoint with 10% critical range	✓	✓	✓	✓	✓	✓
Proposed checkpoint with 25% critical range	✓	✓	✓	✓	✓	✓
Proposed checkpoint with 40% critical range	✓	✓	✓	✓	✓	✓

Table 6.2: Data Gathered

6.6 Calculating the Checkpointing Intervals using Young's First Order Approximation.

To calculate the checkpoint time for the PI and Fire Reduce application based on Young's First Order Approximation (FOA), we needed to find out the number of errors obtained when each application were executed for a period of time as well as the time to take a single checkpoint.

Checkpointing Interval for PI using Young's FOA.

Number of errors in 200 hours of running = 9

Time to perform 1 checkpoint ≈ 3.1 seconds.

Using the first order approximation, the following result was calculated:

1. The mean time between failures T_f = (200*3600)/9 = 80000s
2. The optimum checkpoint interval T_c = sqrt(2*3.1*80000) ≈ 701s

Checkpointing Interval for Fire Reduce using Young's FOA

Number of errors in 150 hours of running = 11.

Time to perform 1 checkpoint ≈ 2.9 seconds.

Using the first order approximation, the following result was calculated:

T_f = (150*3600)/11 ≈ 49090.91s.

T_c = sqrt(2*2.9*49090.91) ≈ 535s

In cases where T_s is not negligible as compared to T_f, Young's second order approximation can be used to determine the checkpointing intervals. The formula is as follows:

$$T_c = \sqrt{2T_sT_f - T_s^2}. \quad (38)$$

Where

T_s is the time required to save information at a checkpoint (Ts).

T_f is the mean time between failures (T_f)

For testing PI and Fire_Reduce, Young's First Order Approximation was used because T_s is very small as compared to T_f. For example in the case of fire reduce, T_s is approximately 2.9 seconds and T_f is approximately 49090.91 seconds.

Using the First Order Approximation, the checkpointing value is:

$T_c \approx$ 534.517 ≈ 535s

Using the second order approximation, the value is:

$$T_c = \sqrt{(2*2.9*49090.91) - (2.9^2)} \approx 534.509 \approx 535s. \quad (39)$$

Similarly, for PI, T_s (3.1s) is very small as compared T_f (80000). The difference between the results obtained using Young's First Order and Second Order Approximation is negligible.

It should be noted that the proposed solution uses Young's First Order of Approximation to calculate the best possible regular checkpoint interval for a given application. If there are better solutions to determine the best possible regular checkpointing interval, they can be used instead.

6. 7 Tests

This section explains how the tests were carried out and the results obtained.

6.7.1 Test Result for PI

Figure 6.2 below shows the execution time line of the PI application. The checkpoint time obtained using the Young's FOA is 701 seconds and there are 6 natural synchronisation points along the execution line at approximately 203, 353, 608, 962, 1131, 1402 and 1556 seconds.

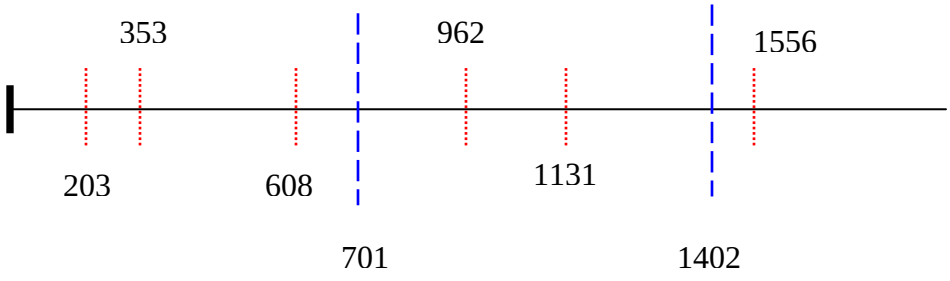

Figure 6.2: PI execution model

Without Failure

Execution without failure		
	Execution time	**Comments**
No checkpoints	1899	
	1845	
	1901	
	1946	Ignored
	1860	
	1895	
	1922	Ignored
	1903	
	1897	
	1864	
	1901	
Average time	**1885**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint every 200 seconds.	1912	
	1925	
	1922	
	1867	Ignored
	1931	
	1840	Ignored
	1916	
	1926	
	1826	Ignored
Average time	**1922**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint at NSP.	1912	
	1884	
	1889	
	1934	Ignored
	1925	Ignored
	1889	
	1899	
	1913	
	1905	
Average time	**1899**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint at Young's FOA Checkpoint Interval.	1905	
	1782	Ignored
	1905	
	1897	
	1948	
	1910	
	1902	
	1882	Ignored
Average time	**1900**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpointing Mechanism" with a critical range of 10% of Young's Checkpoint value	1904	
	1880	
	1883	
	1885	
	1842	Ignored
	1895	
	1900	
	1892	
	1913	
	1909	
Average time	**1896**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpointing Mechanism" with a critical range of 25% of Young's Checkpoint value	1879	
	1886	
	1787	Ignored
	1887	
	1838	Ignored
	1873	
	1905	
	1908	
	1902	
Average time	**1891**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpointing Mechanism" with a critical range of 40% of Young's Checkpoint value	1944	Ignored
	1891	
	1859	
	1901	
	1903	
	1893	
	1728	Ignored
	1903	
Average time	**1892**	

Table 6.3: Execution without failure for PI application

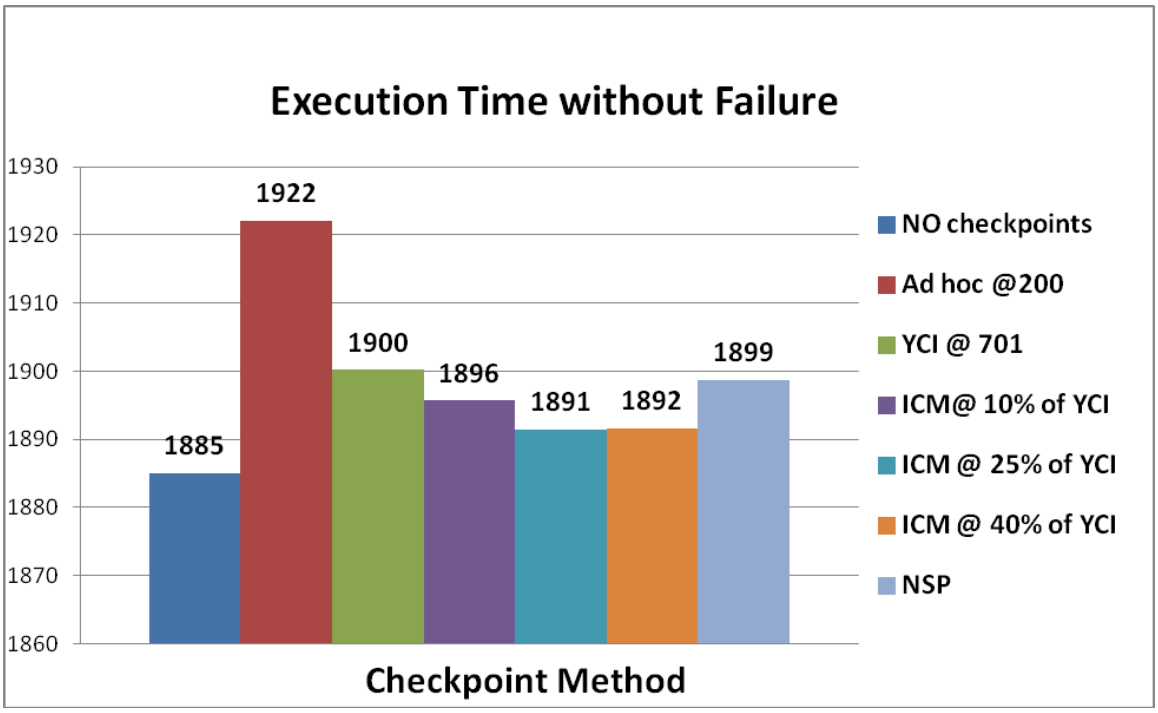

Figure 6.3: Execution Time without Failure for PI

The result in figure 6.3 shows:

1. Execution time without checkpointing: Time (1885s)
2. Checkpoint at ad hoc regular intervals (every 200s) : Checkpoints files (9), Time (1922 s)
3. Checkpoint at Young's Checkpoint Intervals (YCI@701) : Checkpoints files (2), Time (1900s)
4. Checkpoint at Natural Synchronisation Points (NSP): Checkpoint files (6), Time (1899).
5. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 10% of Young's checkpoint interval (ICM @ 10% of YCI): Checkpoints files (2), Time (1896 s).
6. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 25% of Young's checkpoint interval (ICM @ 25% of YCI): Checkpoints files (2), Time (1891 s)

7. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 40% of Young's checkpoint interval (ICM @ 40% of YCI): Checkpoints files (2), Time (1892 s)

With Failure

Execution with failure		
	Execution time	**Comments**
Taking checkpoint every 200 seconds.	2195	Ignored
	1987	Ignored
	2075	
	2084	
	2090	
	2093	
	2088	
	2090	
	2080	
Average time	**2086**	

Execution with failure		
	Execution time	**Comments**
Taking checkpoint at NSP.	1873	Ignored
	2040	
	2043	
	2057	
	2040	
	2049	
	2035	
	1846	Ignored
	2061	
Average time	**2046**	

Execution with failure		
	Execution time	**Comments**
Taking checkpoint at Young's FOA Checkpoint Interval.	2063	
	2052	
	2064	
	2066	
	2055	
	2089	
	2089	
	2056	
	2055	
	2130	Ignored
Average time	**2065**	

Execution with failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of Young's Checkpoint value	2040	
	1956	Ignored
	1980	Ignored
	2081	
	2025	
	2029	
	2024	
	2080	
	2022	
	1901	Ignored
Average time	**2043**	

Execution with failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value	2031	
	2065	
	2030	
	2044	
	2020	
	2040	
	2045	
	1999	Ignored
	1990	Ignored
Average time	**2039**	

Execution with failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value	2054	
	2025	
	2044	
	2045	
	2030	
	2056	
	2045	
	2039	
	2024	
Average time	**2040**	

Table 6.4: Execution with failure for PI application

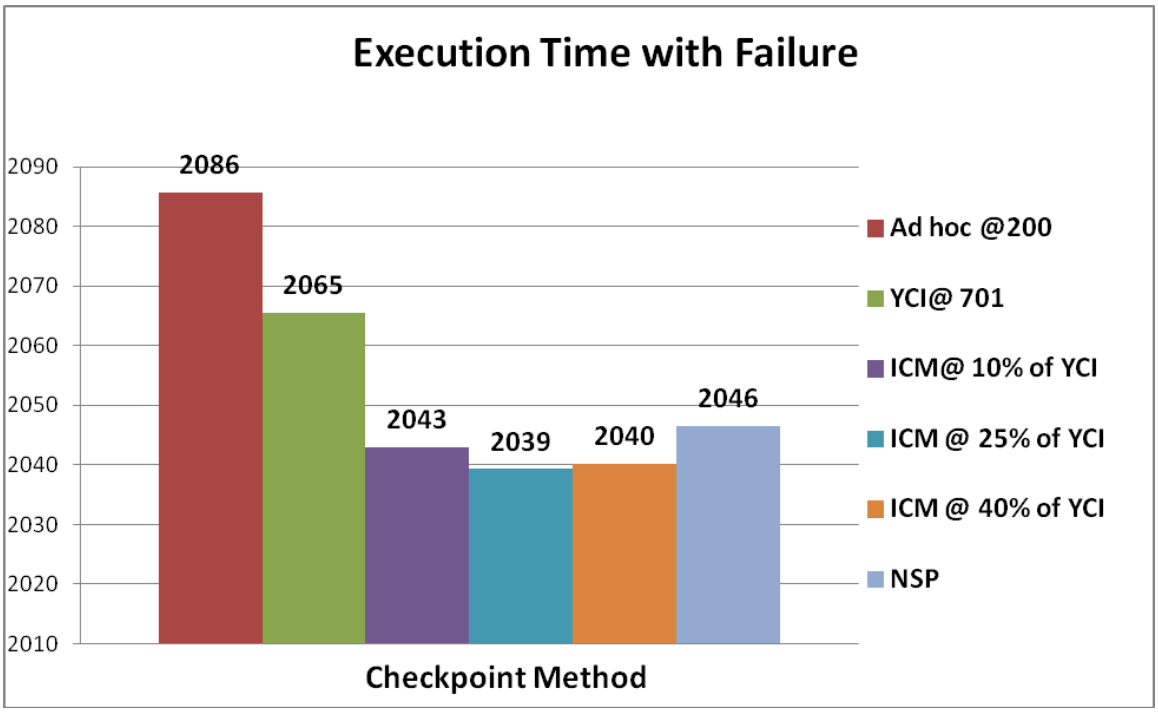

Figure 6.4: Execution Time with Failure for PI

The result in figure 6.4 shows:

1. Checkpoint at ad hoc regular intervals (every 200 s) : Checkpoints files (9), Time (2086 s)
2. Checkpoint at Young's Checkpoint Intervals (YCI@701) : Checkpoints files (2), Time (2065 s)
3. Checkpoint at Natural Synchronisation Points (NSP) : Checkpoint files (6), Time (2046s).
4. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 10% of Young's checkpoint interval (ICM @ 10% of YCI): Checkpoints files (2), Time (2043 s)
5. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 25% of Young's checkpoint interval (ICM @ 25% of YCI): Checkpoints files (2), Time (2039 s)

6. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 40% of Young's checkpoint interval (ICM @ 40% of YCI): Checkpoints files (2), Time (2040 s)

Recovery Time

Table 6.5 and Figure 6.5 below show the recovery time for each method. The result shows that it is much faster to recover checkpoints taken at natural synchronisation points. This is because its does not require restoring in-transit messages and their coordination to restart the application.

Recovery Time			
	Execution Time with failure	**Execution time without failure**	**Recovery Time**
Taking checkpoint every 200 seconds.	2086	1922	**164**
Taking checkpoint at NSP.	2046	1899	**147**
Taking checkpoint at Young's FOA Checkpoint Interval.	2065	1900	**165**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of Young's Checkpoint value	2043	1896	**147**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value.	2039	1891	**148**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value.	2040	1892	**148**

Table 6.5: Recovery Time for PI application

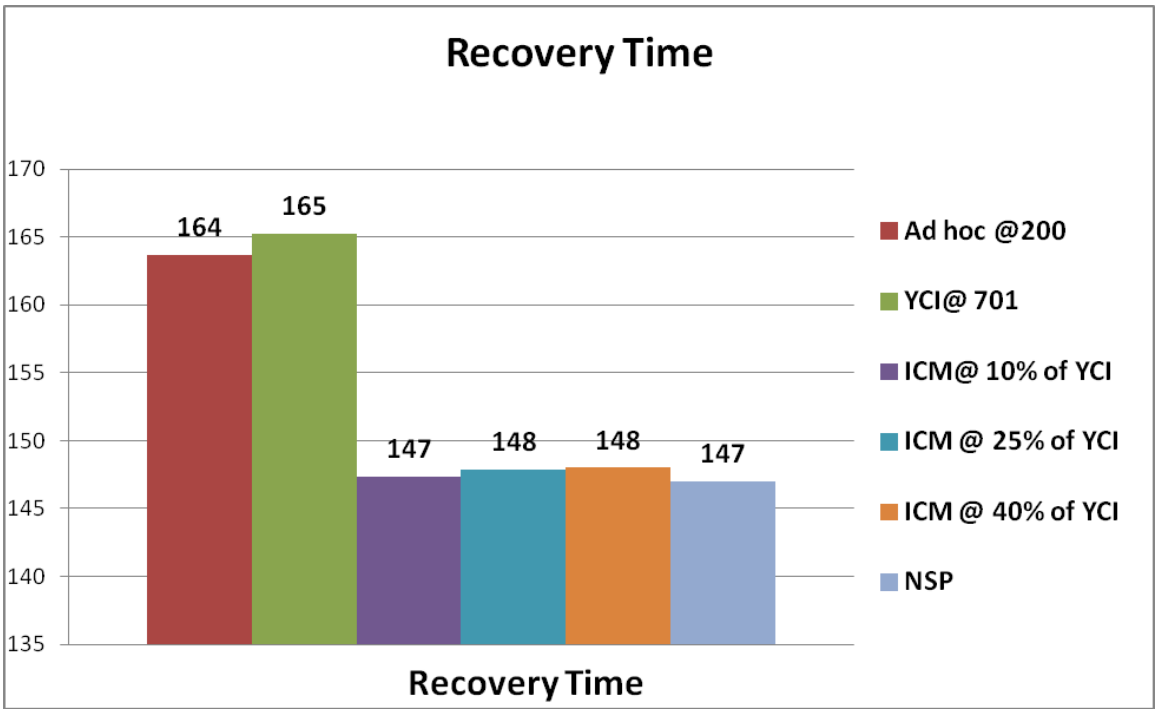

Figure 6.5: Recovery Time for PI

Percentage increase in Execution time

The percentage change in execution time is shown in Figure 6.6 below. The best result is for checkpointing using the "Improved Checkpoint Mechanism" with critical regions of 25% and 40% of Young's Checkpoint Interval. This is mainly because both checkpoints are taken at natural synchronisation points resulting in a quicker execution time.

Percentage increase in Execution time		
	Execution Time	**Percentage increase in Execution time**
No checkpoints	1885	
Taking checkpoint every 200 seconds.	1922	((1922-1885)/1885)*100= 1.96
Taking checkpoint at NSP.	1899	((1899-1885)/1885)*100= 0.74
Taking checkpoint at Young's FOA Checkpoint Interval.	1900	((1900-1885)/1885)*100= 0.80
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of Young's Checkpoint value.	1896	((1896-1885)/1885)*100= 0.58
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value.	1891	((1891-1885)/1885)*100= 0.32
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value.	1892	((1892-1885)/1885)*100= 0.37

Table 6.6: Percentage increase in Execution time for PI

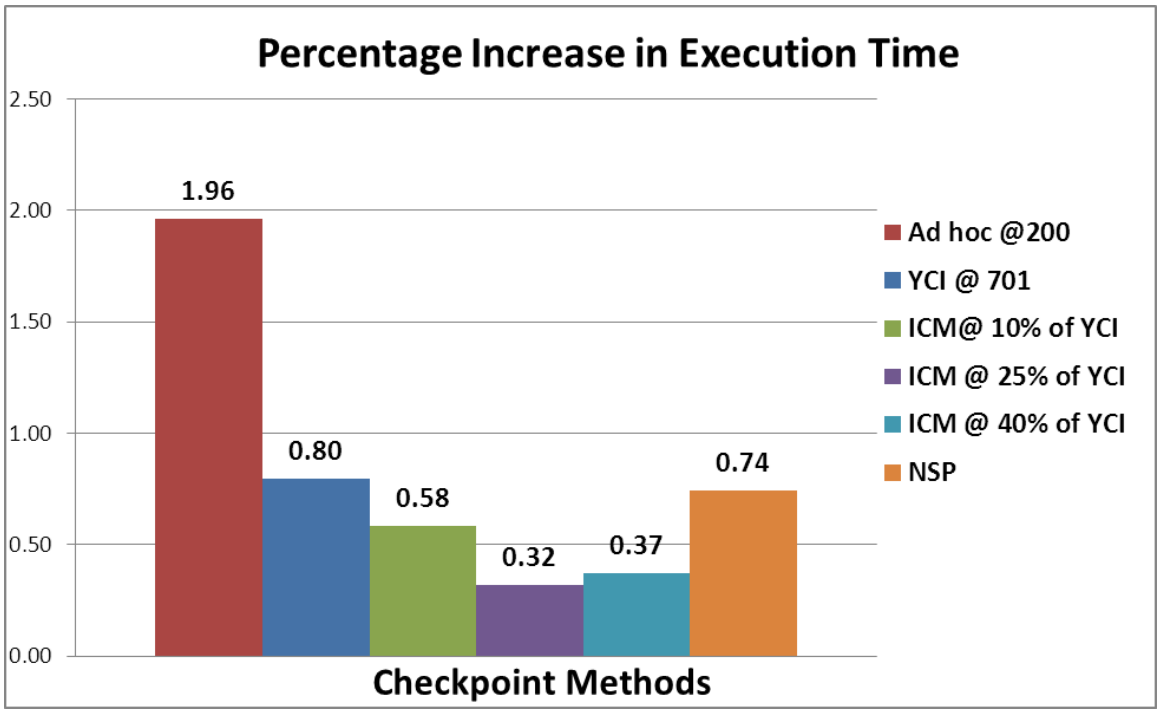

Figure 6.6: Percentage increase in execution time for PI

Percentage change in Time as Compared to Ad hoc Regular Checkpoint

Therefore checkpointing at NSPs improves the performance. Table 6.7 and Figure 6.7 below shows that the most efficient solution would be taking checkpoint when using the "Improved Checkpoint Mechanism" with a critical region of 25% of Young's Checkpoint value (ICM @ 25% of YCI). There is an improvement of more that 83% as compared to the time it takes to execute an application with checkpoint at ad hoc regular intervals.

Percentage change in Time as Compared to Ad hoc Regular Checkpoint		
	Percentage increase in Execution time	**percentage decrease in Time as Compared to reg. chkpt**
Taking checkpoint every 200 seconds.	1.96	(100 + ((1.96-1.96)/1.96)*100) = 100
Taking checkpoint at NSP.	0.74	(100 - ((1.96-0.74)/1.96)*100) = 37.76
Taking checkpoint at Young's FOA Checkpoint Interval.	0.80	(100 - ((1.96-0.8)/1.96)*100) = 40.82
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of Young's Checkpoint value.	0.58	(100 - ((1.96-0.58)/1.96)*100) = 29.59
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value.	0.32	(100 - ((1.96-0.32)/1.96)*100) = 16.33
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value.	0.37	(100 - ((1.96-0.37)/1.96)*100) = 18.88

Table 6.7: Percentage change in Time as Compared to Regular Checkpoint for PI

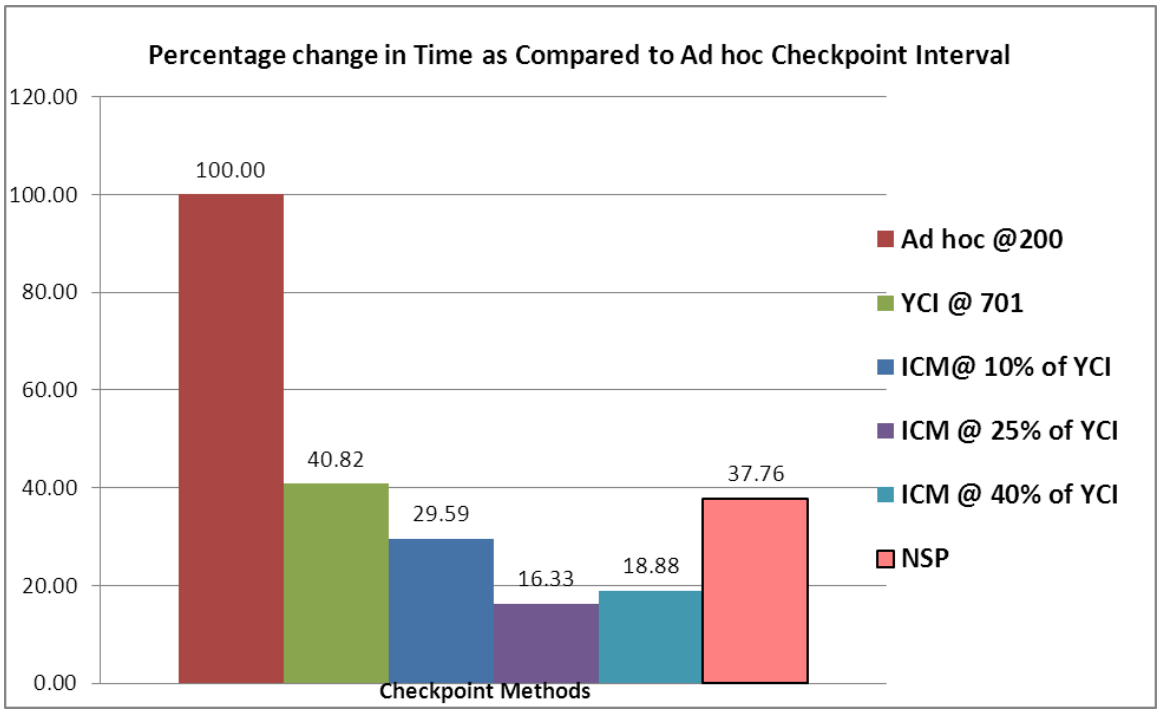

Figure 6.7: Percentage change in Time as Compared to Regular Checkpoint for PI

Average Checkpoint Time		
	Checkpoint taken at (seconds)	**Average checkpoint time**
Taking checkpoint every 200 seconds.	200,400,600,800,1000 ,1200,1400,1600,1800	200
Taking checkpoint at NSP.	208, 353, 608, 962, 1131, 1556	259
Taking checkpoint at Young's FOA Checkpoint Interval.	701, 1402	701
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of Young's Checkpoint value.	631, 1473	736
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value.	605, 1548	774
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value.	421, 1526	763

Table 6.8: Average Checkpoint Time for PI

The checkpoint interval using Young's FOA for this application is 701 seconds. In this case, the best solution is when a checkpoint is taken at 701 seconds. However, the proposed solution does not deviate too much from Young's Checkpoint Interval value as shown in the Figure 6.8 below.

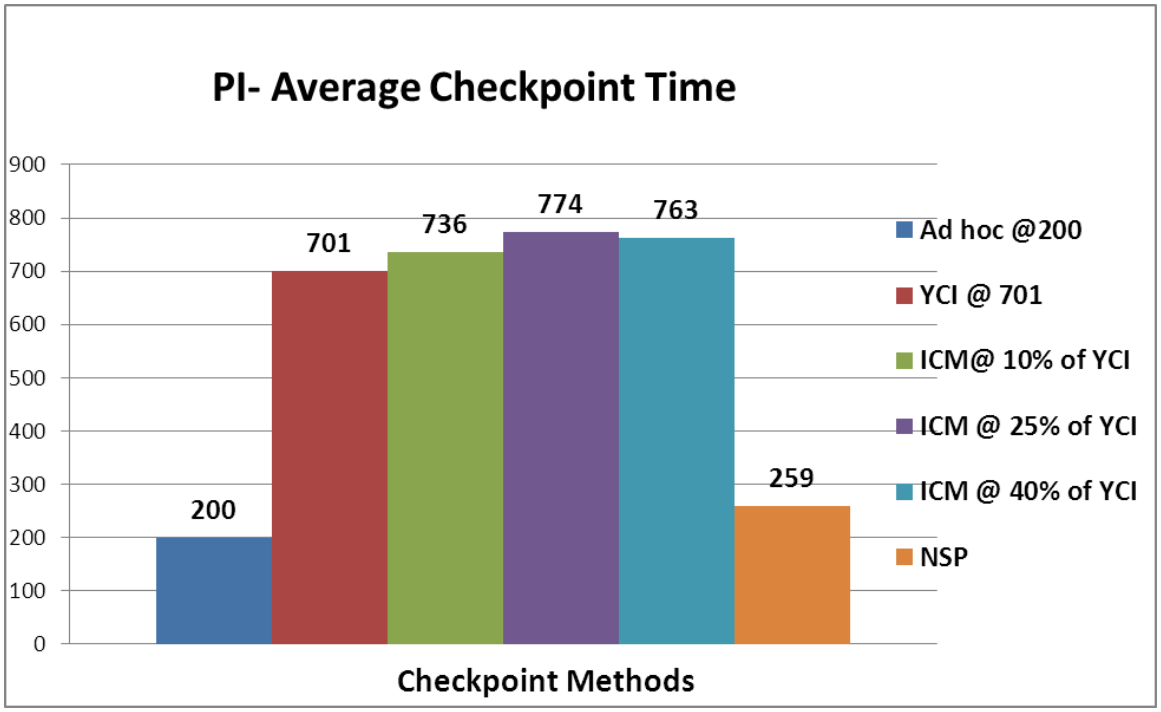

Figure 6.8: Average Checkpoint Time for PI

By further analysing the result shown in the Table 6.8 above, we can conclude that taking checkpoint using the "Improved Checkpoint Mechanism" with a critical region of 10%, 25% and 40% of Young's Checkpoint Interval deviate around 5%, 10% and 8% respectively as compared to the ad hoc checkpoint interval (200s) and checkpoint at NSP which deviates by 72% and 63% respectively as shown on the Figure 6.9 below.

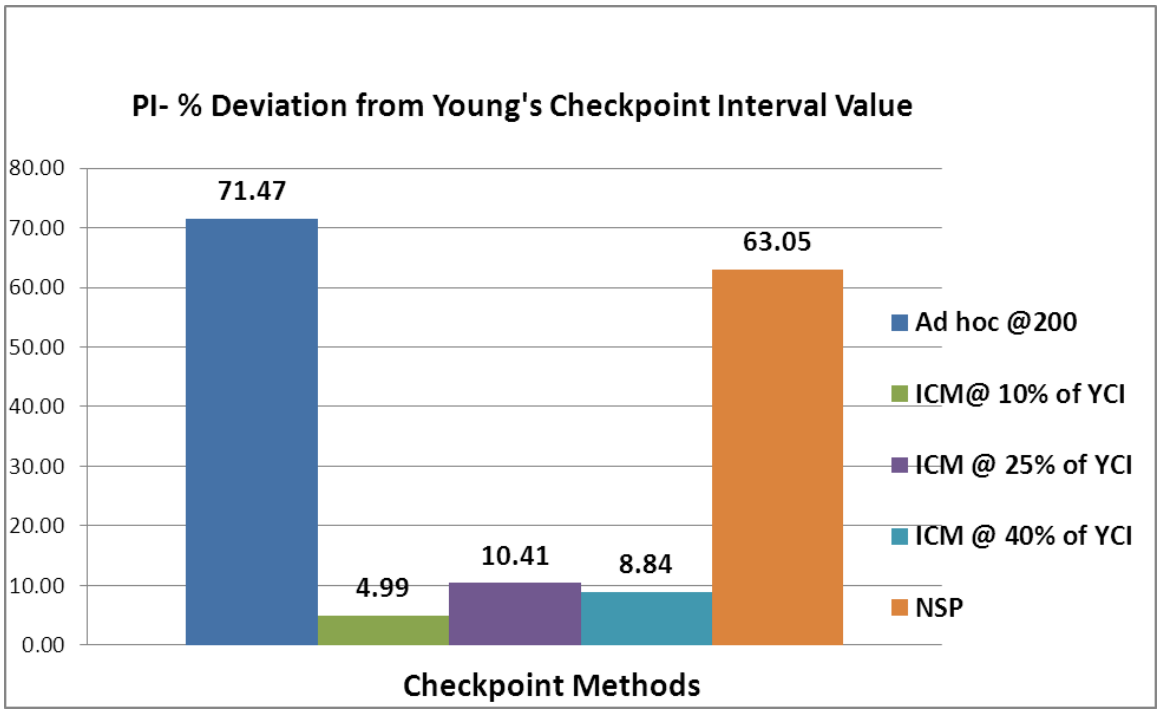

Figure 6.9: Percentage deviation from Young's Checkpoint value for PI

As we saw earlier, there is a decrease in execution time when checkpoints are taken at natural synchronisation points. Therefore if a checkpoint is taken at a NSP rather than a forced checkpoint, the performance is better.

Figure 6.10 shows that the bigger the critical interval, the higher is the chance of getting a NSP for checkpointing.

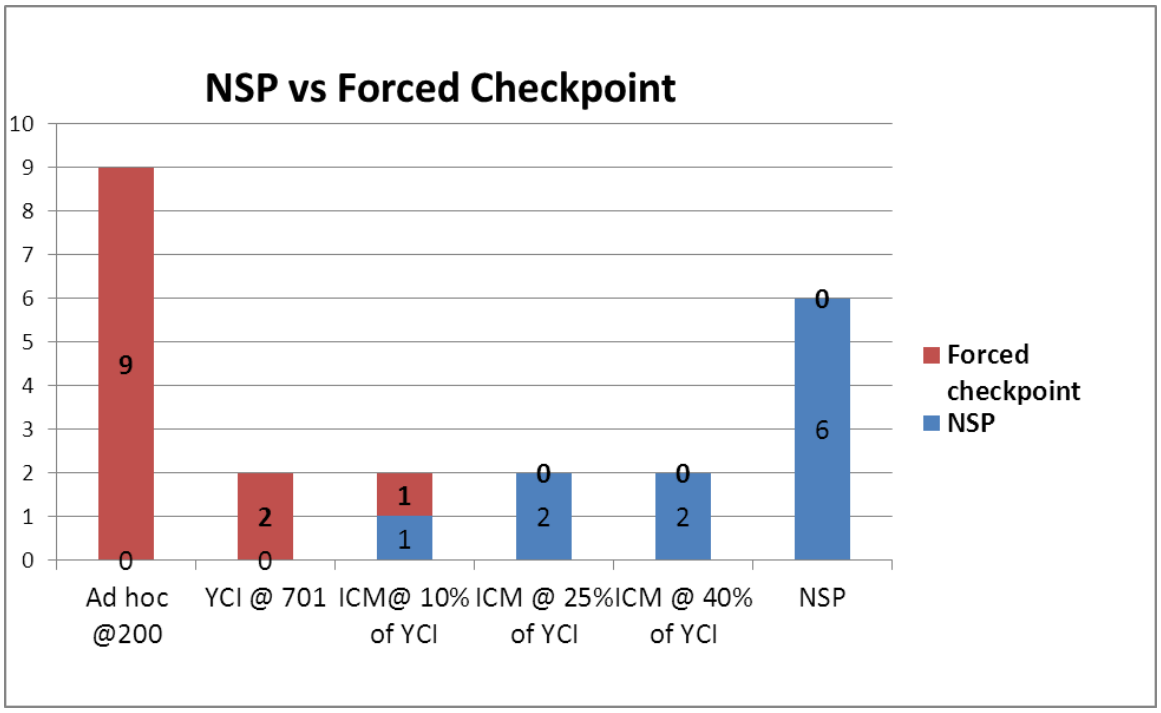

Figure 6.10: NSP vs. forced Checkpoint for PI

6.7.2 Test Results for Fire Reduce

Fire Reduce - Execution without failure		
	Execution time	**Comments**
No checkpoints	1100	
	1120	
	1067	Ignored
	1056	Ignored
	1081	
	1086	
	1112	
	1101	
	1103	
	1160	Ignored
Average time	**1100**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint every 100 seconds.	1139	
	1133	
	1086	Ignored
	1089	Ignored
	1180	
	1180	
	1150	
	1143	
	1144	
Average time	**1153**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint at Young's FOA Checkpoint Interval.	1141	
	1135	
	1122	
	1166	
	1130	
	1121	
	1116	
	1133	
	1046	Ignored
Average time	**1133**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of Young's Checkpoint value	1173	Ignored
	1189	Ignored
	1102	
	1121	
	1096	
	2002	Ignored
	1126	
	1106	
	1116	
	1101	
Average time	**1110**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value	1120	
	1106	
	1111	
	1082	
	1106	
	1170	Ignored
	1060	
	1175	Ignored
	1176	Ignored
	1173	Ignored
Average time	**1108**	

Execution without failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value	1181	Ignored
	1111	
	1127	
	1099	
	1090	
	1112	
	1106	
	1115	
	1170	Ignored
Average time	**1109**	

Table 6.9: Execution without failure for Fire Reduce

Without Failure

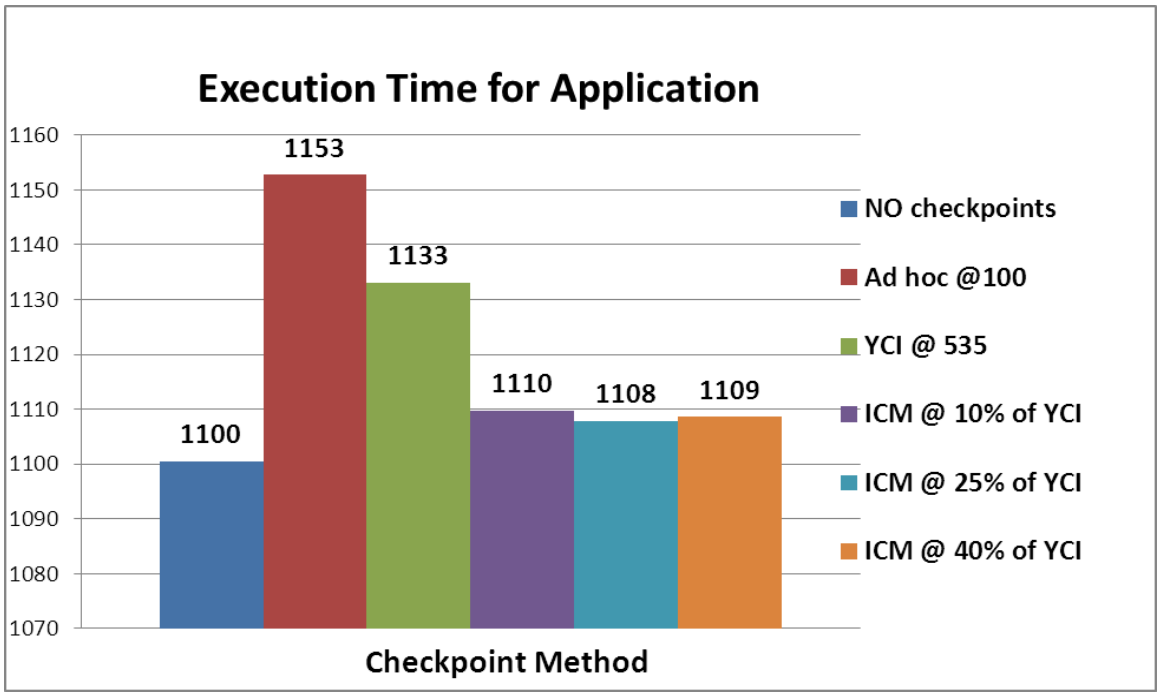

Figure 6.11: Execution Time without Failure for Fire Reduce

The result in Figure 6.11 shows:

1. Execution time without checkpointing: Time (1100s)
2. Checkpoint at ad hoc regular intervals (every 100s) : Checkpoints files (11), Time (1153 s)
3. Checkpoint at Young's Checkpoint Intervals (every 535s): Checkpoints files(2), Time (1133 s).
4. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 10% of Young's checkpoint interval (ICM @ 10% of YCI): Checkpoints files (2), Time (1110 s)
5. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 25% of Young's checkpoint interval (ICM @ 25% of YCI): Checkpoints files (2), Time (1108 s)
6. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 40% of Young's checkpoint interval (ICM @ 40% of YCI): Checkpoints files (2), Time (1109 s)

With Failure

Execution with failure		
Taking checkpoint every 100 seconds.	**Execution time**	**Comments**
	1270	
	1240	
	1248	
	1196	Ignored
	1235	
	1270	
	1257	
	1235	
	1243	
	1260	
Average time	**1251**	

Execution with failure		
Taking checkpoint at Young's FOA Checkpoint Interval.	**Execution time**	**Comments**
	1230	
	1248	
	1224	
	1241	
	1212	
	1136	Ignored
	1220	
	1175	Ignored
	1251	
	1244	
Average time	**1234**	

Execution with failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of Young's Checkpoint value	1225	Ignored
	1202	
	1118	Ignored
	1201	
	1211	
	1180	
	1150	
	1216	
	1172	
	1224	Ignored
Average time	**1190**	

Execution with failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value	1170	
	1205	
	1192	
	1165	
	1180	
	1204	
	1170	
	1227	Ignored
	1229	Ignored
	1198	
Average time	**1186**	

Execution with failure		
	Execution time	**Comments**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value	1207	
	1118	Ignored
	1162	
	1205	
	1117	Ignored
	1197	
	1170	
	1188	
	1175	
Average time	**1186**	

Table 6.10: Execution with failure for fire reduce

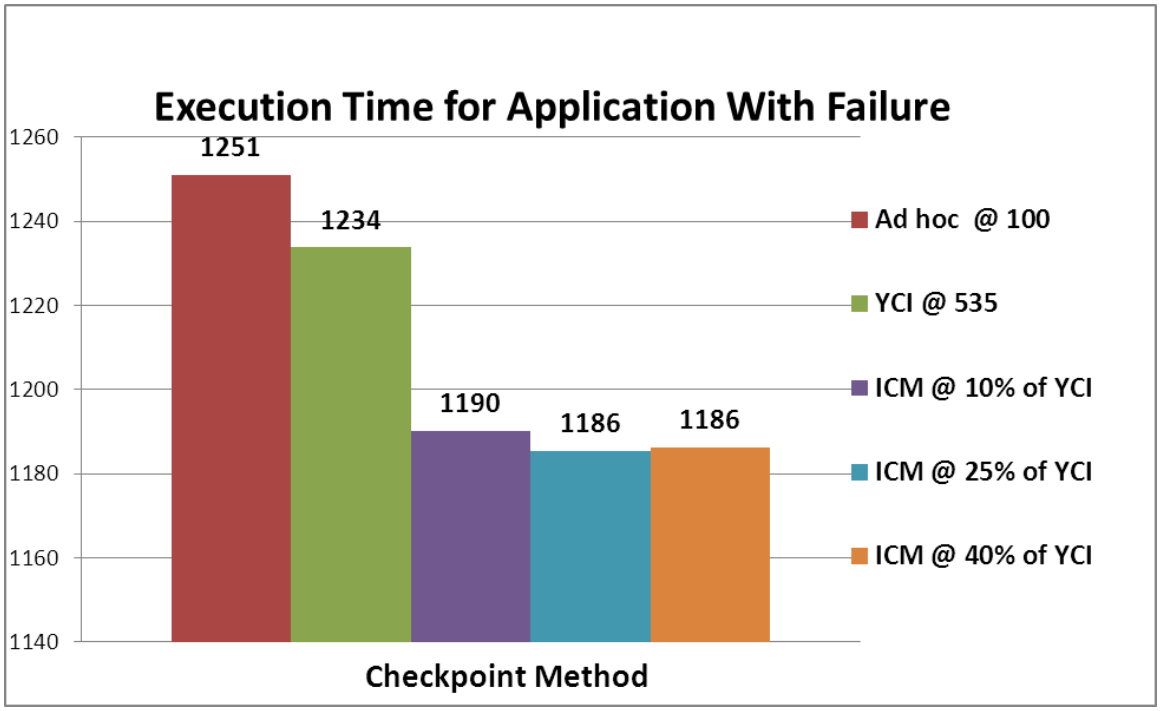

Figure 6.12: Execution Time with Failure for Fire Reduce

The result in Figure 6.12 shows:

1. Checkpoint at ad hoc regular intervals (every 100 s): Checkpoints files (10), Time (1251 s)
2. Checkpoint at Young's Checkpoint Intervals (every 535s): Checkpoints files (2), Time (1234 s)
3. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 10% of Young's checkpoint interval (ICM @ 10% of YCI): Checkpoints files (2. Both at NSP), Time (1190 s)
4. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 25% of Young's checkpoint interval (ICM @ 25% of YCI): Checkpoints files (2. Both at NSP), Time (1186 s)
5. Checkpoint using "Improved Checkpoint Mechanism" with a critical range of 40% of Young's checkpoint interval (ICM @ 40% of YCI): Checkpoints files (2. Both at NSP), Time (1186 s)

Recovery Time

Table 6.11 and Figure 6.13 below show the recovery time for each method. The result shows that it is faster to recover checkpoints taken at natural synchronisation points. This is because it does not require restoring in-transit messages and their coordination to restart the application.

Recovery Time			
	Execution Time with failure	**Execution time without failure**	**Recovery Time**
Taking checkpoint every 100 seconds.	1251	1153	**98**
Taking checkpoint at Young's FOA Checkpoint Interval.	1234	1133	**101**
Taking checkpoint at using the "Improved Checkpointing Mechanism" with a critical range of 10% of Young's Checkpoint value	1190	1110	**80**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value	1186	1108	**78**
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value	1186	1109	**77**

Table 6.11: Recovery time for fire reduce

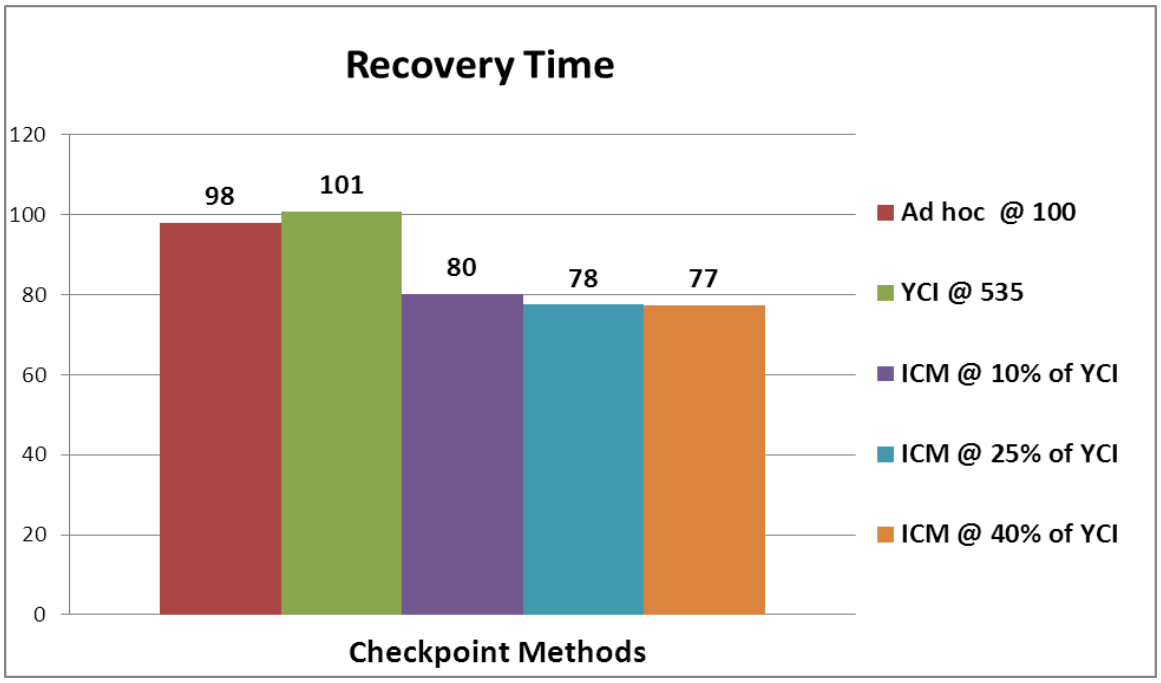

Figure 6.13: Recovery Time for Fire Reduce

Percentage increase in Execution time

Percentage increase in Execution time		
	Execution Time	**Percentage increase in Execution time**
No checkpoints	1100	
Taking checkpoint every 100 seconds.	1153	((1153-1100)/1100)*100= 4.75
Taking checkpoint at Young's FOA Checkpoint Interval.	1133	((1133-1100)/ 1100)*100= 2.96
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of the optimal time.	1110	((1110-1100)/ 1100)*100= 0.84
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value	1108	((1108-1100)/ 1100)*100= 0.65
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value	1109	((1109-1100)/ 1100)*100= 0.74

Table 6.12: Percentage increase in Execution time for Fire Reduce

The percentage change in execution time is shown in Table 6.12 above and Figure 6.14 below. The best results are obtained when checkpoints are taken at the "Improved Checkpointing Mechanism" with a critical region of 10%, 25% and 40% of Young's Checkpoint Interval (ICM @ 25% of YCI). This is mainly because both checkpoints are taken at Natural Synchronisation Points resulting in a quicker execution time.

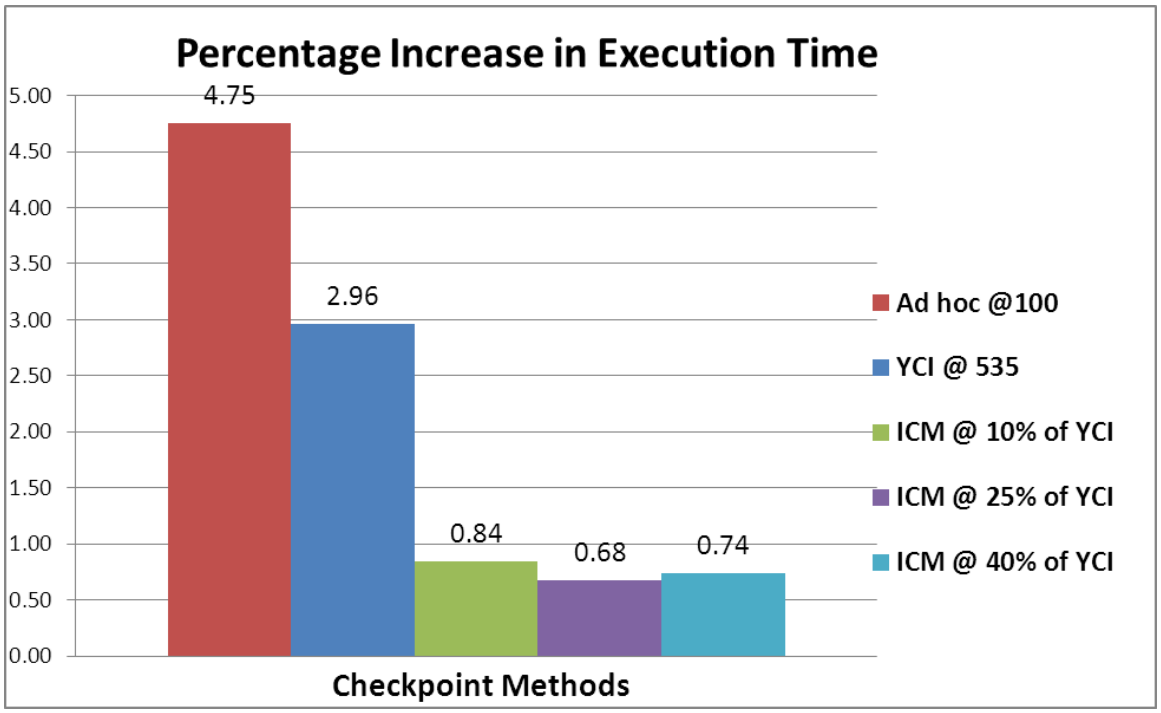

Figure 6.14: Percentage increase in Execution time for Fire Reduce

Percentage change in Time as Compared to Ad hoc Regular Checkpoint

Therefore checkpointing at NSP improves the performance. The Table 6.13 and Figure 6.15 below show that the most efficient solution would be taking checkpoint using the "Improved Checkpoint Mechanism" with a critical region of 25% of Young's Checkpoint Interval (ICM @ 25% of YCI). There is an improvement of more that 86 % as compared to the time it takes to execute an application with checkpoint at ad hoc regular intervals. In this example, the result is very good in all the three critical range options.

Percentage change in Time as Compared to Ad hoc Regular Checkpoint		
	Percentage increase in Execution time	**percentage decrease in Time as Compared to reg chkpt**
Taking checkpoint every 100 seconds.	4.75	(100 + ((4.75-4.75)/ 4.75)*100) = 100
Taking checkpoint at Young's FOA Checkpoint Interval.	2.96	(100 - ((4.75-2.96)/ 4.75)*100) = 62.30
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of Young's Checkpoint value	0.84	(100 - ((4.75-0.84)/ 4.75)*100) = 17.76
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value.	0.65	(100 - ((4.75-0.65)/ 4.75)*100) = 14.71
Taking checkpoint at using "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value.	0.74	(100 - ((4.75-0.74)/ 4.75)*100) = 15.57

Table 6.13: Percentage change in Time as Compared to Ad hoc Regular Checkpoint for fire reduce

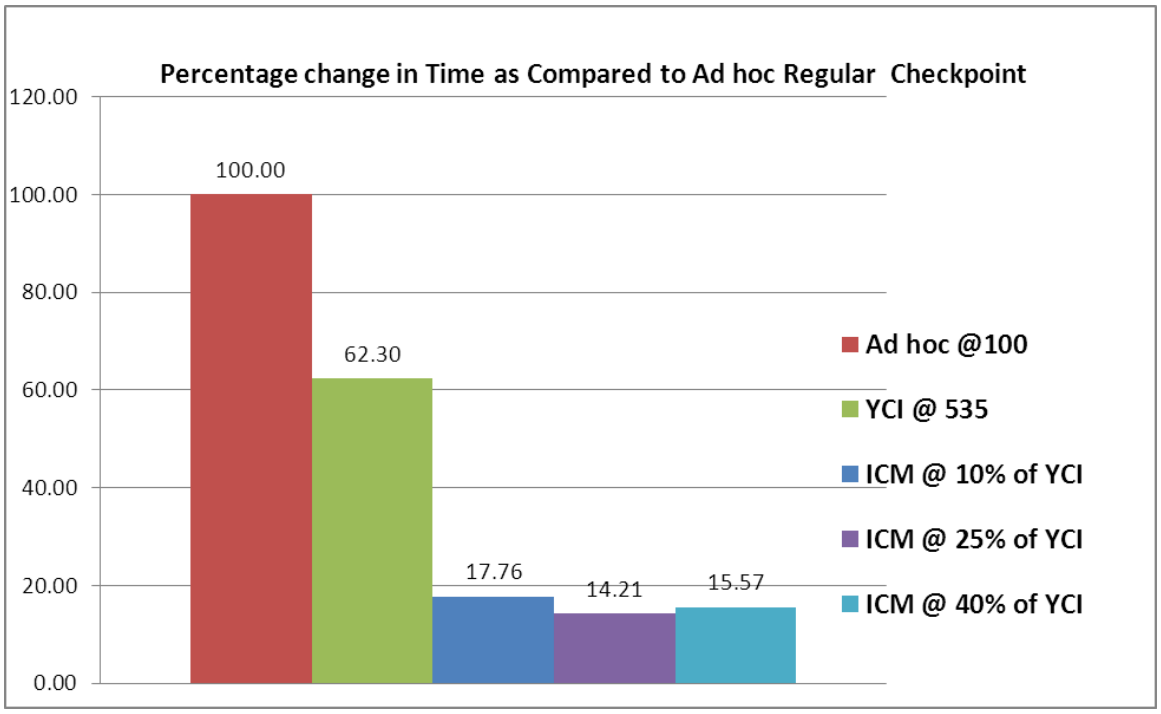

Figure 6.15: Percentage Change in Time as compared to Ad hoc Reg. Checkpoint for Fire Reduce

Average Checkpoint Time

The checkpoint time using Young's FOA for this application is 535 seconds. In this case, the best solution is when a checkpoint is taken every 535 seconds. However, the proposed solution does not deviate too much from Young's FOA Checkpoint Interval value. The lowest range while checkpointing with the proposed solution is 428 seconds as show in Table 6.14 and Figure 6.16. Figure 6.17 below shows that results obtained when checkpoints are taken using the "Improved Checkpoint Mechanism" with a critical region of 10%, 25% and 40% of Young's Checkpoint Interval (ICM @ 25% of YCI) deviate around 5%, 13% and 20% respectively as compared to ad hoc regular checkpoint which deviates by 81%. Taking checkpoint using a critical range of 10% of Young's Checkpoint Interval looks the best solution. However, the risk of missing a NSP in that region is bigger.

Average Checkpoint Time		
	Checkpoint taken at (seconds)	**Average checkpoint time**
Taking checkpoint every 100 seconds.	100,200,300,400, 500, 600, 700, 800, 900,1000, 1100	100
Taking checkpoint at Young's FOA Checkpoint Interval.	535, 1070	535
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 10% of Young's Checkpoint value.	482, 1018	509
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 25% of Young's Checkpoint value.	403, 938	469
Taking checkpoint at using the "Improved Checkpoint Mechanism" with a critical range of 40% of Young's Checkpoint value.	322, 857	428

Table 6.14: Average Checkpoint Time for Fire Reduce

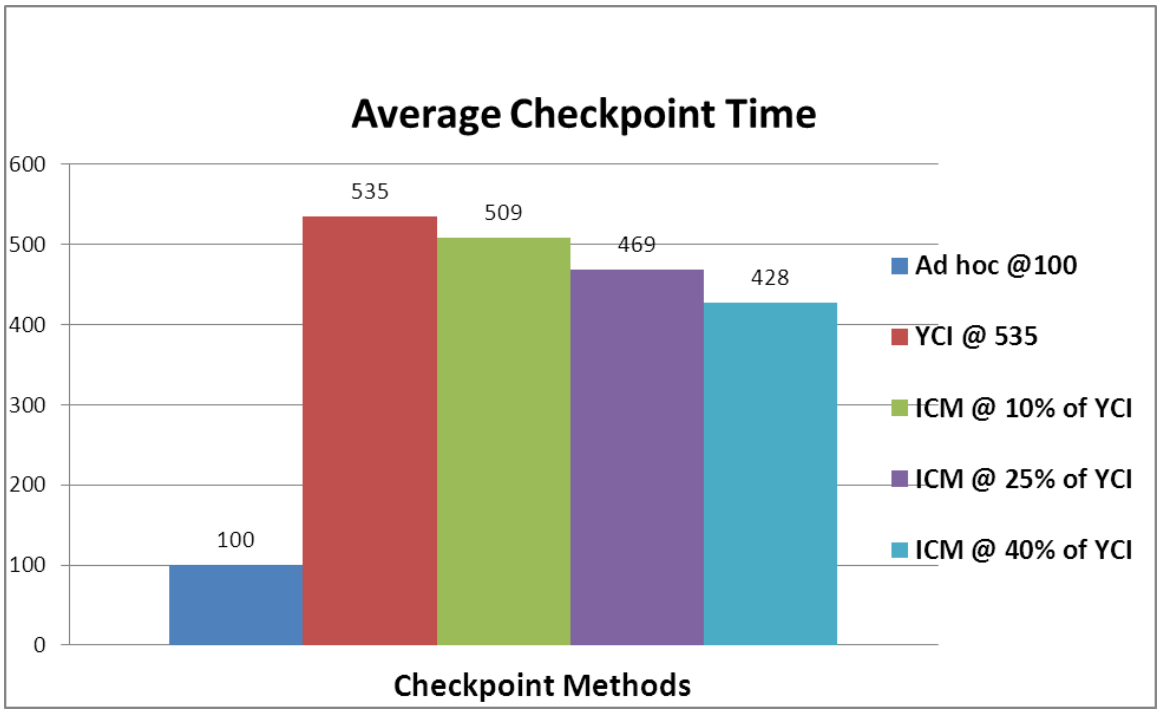

Figure 6.16: Average checkpoint Time for Fire Reduce

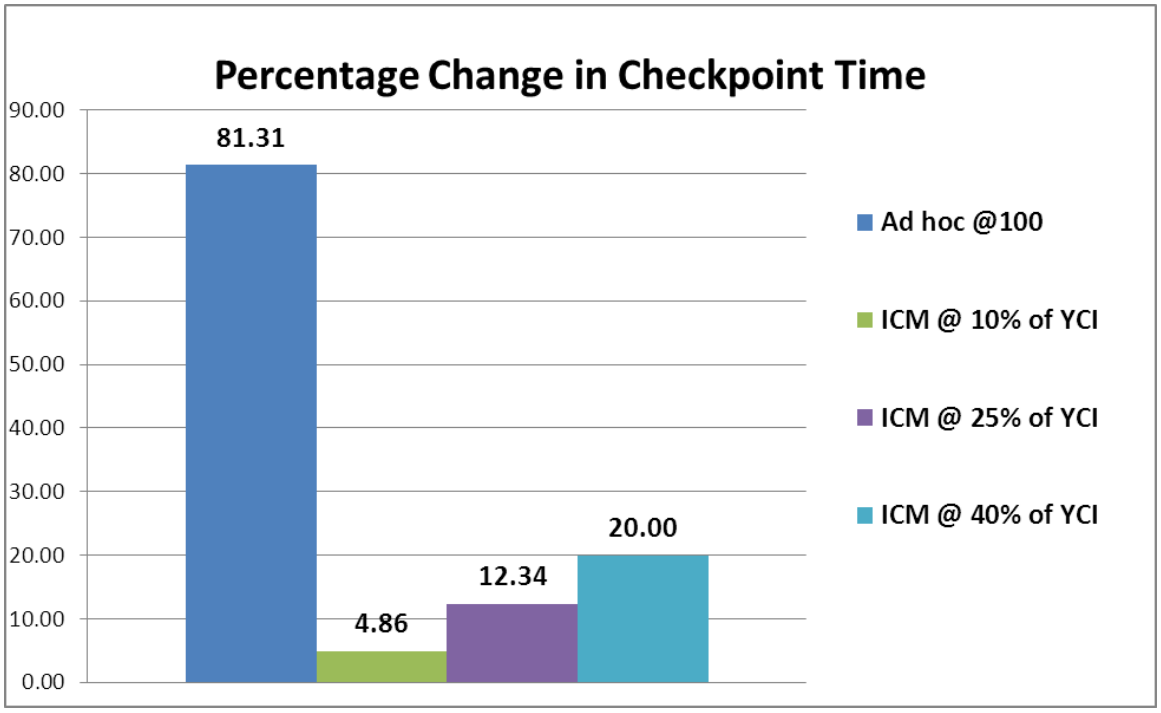

Figure 6.17: Percentage deviation from Young's Checkpoint value for Fire Reduce

For the experimental tests it can be concluded that there is a decrease in execution time when checkpoints are taken at Natural Synchronisation Points. Therefore if a checkpoint is taken at a NSP rather than a forced checkpoint, the performance is better.

Figure 6.18 shows that the bigger the critical interval, the higher is the chance of getting a NSP for checkpointing. In this example, we were able to get a NSP in all the three critical regions selected.

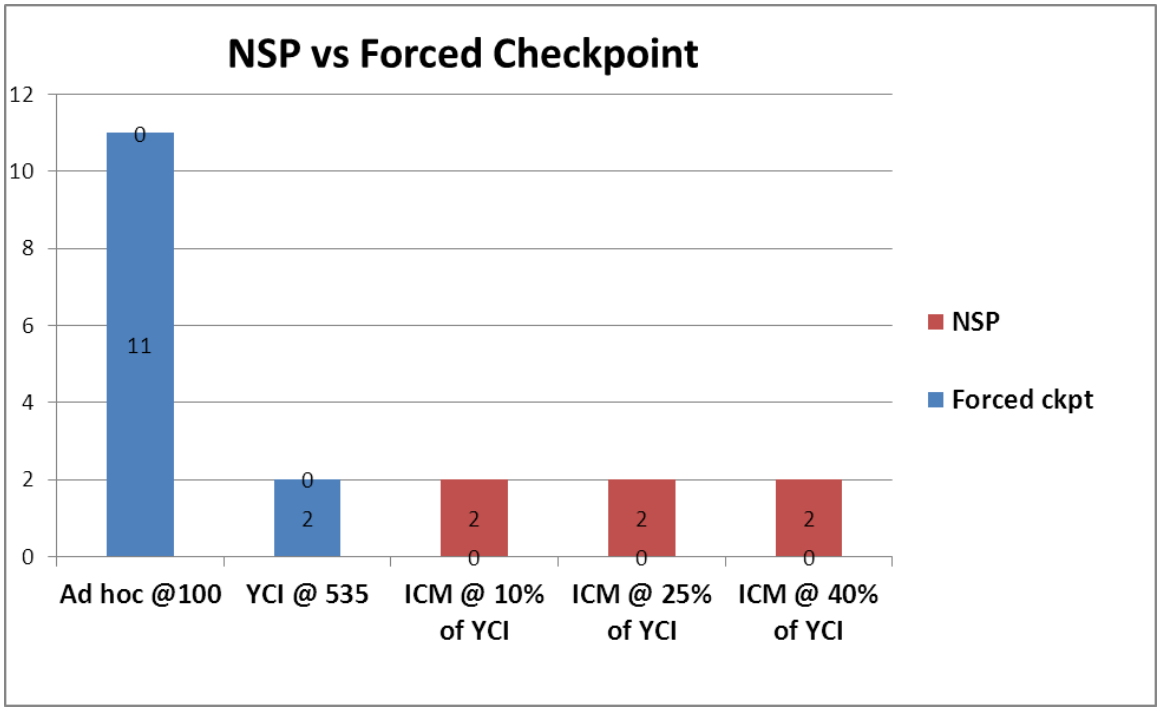

Figure 6.18: NSP vs. Forced Checkpoint for Fire Reduce

6.7.3 Comparing results between the PI and Fire Reduce applications

In the PI application, the occurrence of NSP varied along the time line and there were situations where we had to force a checkpoint as there was no occurrence of any NSPs within a critical region.

In the “Fire Reduce” application, test regarding taking checkpoints at natural synchronisation points was not carried as the occurrence of natural synchronisation points were too quick and this caused the application to fail a few times.
Irrespective of the difference, both applications have shown that the proposed solution has improved the execution time.

Execution time without failure

In both cases the best execution time was achieved when the critical region was at a range of 25% of Young’s Checkpoint value.
The execution time for PI application is 1891 seconds as compared to 1992 seconds if the proposed solution was not used.
The execution time for Fire Reduce application is 1100 seconds as compared to 1153 seconds if the proposed solution was not used.

Execution time with failure

In this case as well, the both applications achieve the best result when the critical region is in a range of 25% of Young’s Checkpoint value.
The execution time for PI application is 2039 seconds as compared to 2086 seconds if the proposed solution was not used.
The execution time for Fire Reduce is 1186 seconds as compared to 1251 seconds if the proposed solution was not used.

Recovery time

Under the proposed checkpointing solution both the PI and Fire reduce showed that the recovery time is better.
The PI application takes 147 seconds to recover as compare to 165 seconds if application was checkpointed at Young’s checkpoint intervals only and 164 seconds if application was checkpointed using a user defined checkpointing interval.
The Fire Reduce application takes 77 seconds to recover as compare to 101 seconds if application was checkpointed at Young’s Checkpoint Intervals only and 98 seconds if application was checkpointed using a user defined checkpointing interval.

Average Checkpoint Time

Both applications show that the average checkpointing time is not too far from Young's Checkpoint value.

For the PI application, the average checkpointing time is, in the worse case, an increase of 10% from Young's Checkpoint time and at best an increase of 5 % from the Young's Checkpoint time.

For the PI application, the average checkpointing time is in the worse case an increase of 20% from Young's Checkpoint value and at best an increase of 4 % from Young's Checkpoint value.

So, both solutions show a good improvement in the execution time as compared to taking checkpoint at regular interval or at Young's checkpointing interval.

6.8 Chapter Summary and Conclusions

In the test carried of "Fire Reduce" application, checkpoints at Natural Synchronisation Points where not taken because there is a natural synchronisation point at the end of the main loop and this loop is executed in repeatedly until the programs end.

From the sets of test performed, we can definitely confirm that the proposed solution does improve the checkpointing performance. Using the "Improved Checkpoint Mechanism", the checkpointing time is quite small hence the improvement in execution time is not significant. However, the longer the checkpointing times for applications, the more significant will be the improvement.

The test also confirmed that checkpoint taken at Natural Synchronisation Point is better than forcing a checkpoint which requires coordination among the processes.

The ideal critical range seems to be at twenty-five percent of the Young's Checkpoint Interval.

Therefore, based on the experimental results, it can be concluded that the proposed solution with a critical range of twenty-five percent of Young's checkpoint interval will definitely provide a better checkpointing solution than existing MPI checkpointing solutions.

Chapter 7

7 Conclusions

7.1 Knowledge Contributions and Summary of Thesis

The Grid environment is generic, heterogeneous, and dynamic with lots of unreliable resources making it very exposed to failures. Therefore, it is essential to develop efficient fault tolerant mechanisms to allow users to successfully execute Grid applications. The main objective of this research was to design and implement a checkpointing mechanism for parallel/distributed applications to ensure that applications run reliably in the Grid environment with minimal overhead incurred during the checkpointing and restart process.

Most of the existing checkpointing solutions for Grid applications checkpoint their application at a periodic interval defined by the user. This is not the best solution as the application can end up taking either too many checkpoints or too few checkpoints. Moreover, they do not have features that reduce overhead induced during the checkpoint/restart process. This is a concern mainly when the purpose of the Grid is to provide efficiency in terms of performance.

One of the main challenges was to design and implement a solution that will ensure that parallel/distributed applications can executed efficiently and reliably by defining effective checkpoint intervals and ensuring that the checkpointing and restart process at these intervals incur minimal overhead. Checkpoint overhead is incurred due to the inter process communications that occur among processes. To ensure a consistent checkpoint, the communication layer should be dealt with effectively to prevent lost or orphan messages that may lead to an inconsistent global checkpointing. So, the first objective was to find out methods to reduce the overhead incurred during the checkpointing process. To achieve this goal, MPI applications were analysed to identify points in the application at which checkpoints could be taken. We found that parallel applications

contain regions where no interprocess communications occurs among processes. These regions are known as Natural Synchronisation Points. Additionally, at these points, the size of the global checkpoint state is minimal. Therefore, these points look very promising to checkpoint the application. However, because there is no pattern in the occurrence of natural synchronisation points, we could not rely only on these points for checkpointing.

Therefore, the second objective was to find out how to improve the checkpointing mechanism so that checkpoints are not taken at all natural synchronisations points but at intervals that will ensure the best performance. To achieve this, the First Order Approximation Method proposed by John W. Young was adopted **[43]**. By using a combination of both the Natural Synchronization Points and the First Order Approximation, a novel solution was developed that would improve the checkpointing process of MPI applications. This mechanism was named the "Improved Checkpoint Mechanism". The mechanism allows a checkpoint to be taken at a Natural Synchronization Point close to a Young's Checkpoint Interval within a region called the Critical Region. If no natural synchronisation point is obtained within the Critical Region, a checkpoint is forced as the end of the Critical Region.

Having successfully achieved the second objective, the next challenge was to design a model and write the algorithms based on this novel solution and eventually implement the solution. After successfully designing the model and written the algorithms, the application was implemented. OPENMPI with BLCR was adopted as a starting point mainly because it is a coordinated checkpointing solution which is suitable for the Grid environment. Using OPENMPI/BLCR as a starting point, the proposed solution was successfully implemented. A set of APIs was successfully implemented that were effectively integrated at the application level to achieve the design aims. The novel checkpointing solution was successfully tested and positive outcomes.

The thesis gives a thorough description of a novel solution for checkpointing parallel applications executed in Grid environment. It starts by giving an overview of the research challenges that triggered this research project. It then introduces the Grid and the fault tolerant techniques that are commonly used in the Grid environment such as retrying,

replication and checkpointing before analysis some existing fault tolerant solutions. Through the analysis of these solutions we concluded that checkpointing is most suited for the Grid environment because of its unreliable nature.

Having select checkpointing as the research base, the next steps were to study existing checkpointing mechanisms and solutions to decide which technique is the best suited option for this research. After considering checkpointing techniques such as Operating System checkpointing, Compiler based checkpointing and Application based checkpointing, the different approaches for checkpointing and Recovery were described and existing checkpointing mechanisms were analysed. Through this process, we concluded that coordinated checkpointing was most suited for this research because of its simpler design and recovery characteristics. However, one of the major drawbacks of this solution was the overhead incurred during the checkpointing and restart process. To tackle this shortcoming, a novel solution named the "Improved Checkpoint Mechanism" was developed, which is described in chapter 4. The solution would improve the checkpointing process and minimise overhead incurred during the checkpointing and restart process. Chapter five of the thesis explained the implementation of the proposed solution which is based on the designed model and the algorithms written. A few APIs were implemented and their pseudo codes were written. As explained above, these APIs were successfully integrated at the application level and tests were successfully carried out to confirm the efficiency and reliability of the proposed solution.

7.2 The Experimental Result Conclusions

The purpose of the tests were twofold; one was to check if the proposed solution gives a better result as compared to other checkpointing strategies and second was to find out the best range for the Critical Region.

To test the proposed checkpointing solution, two applications; "PI" and "Fire_Reduce" were used. These two applications were chosen because they are open source and their execution time could be varied to suite the testing requirements. The University of Westminster's Grid was used as the test bed.

The application was tested when executed in a failure free environment and when a failure is induced. Under each condition above, the following tests were carried: running

the application without performing checkpoints; checkpoint the application at an ad hoc regular intervals; checkpoint the application at intervals based on the Young's First Order Approximation; checkpoint the application at natural synchronisation points (NSP) only and checkpoint the application using the proposed solution. When testing the proposed solution, three different sets of test were performed with varying critical ranges (10%, 25% and 40% of the Young's Checkpointing Interval).

The test results showed that the proposed solution gave better results as compared to the other checkpointing strategies. The execution time was faster, the recovery time was better and the checkpoint interval was very close the Young's Checkpoint Intervals. Moreover, the test also confirmed that the best critical range is 25% of Young's Checkpoint Interval.

Therefore we concluded that the proposed checkpointing solution successfully achieved the research aims by providing a better checkpointing option with improved performance.

7.3 Future Work

One important research area associated with proposed solution would be to analyse and implement a solution that will allow to efficiently reduce the size of the checkpointing file. By studying the checkpointing files, a solution can be designed and implemented to ensure that only data needed to successfully restart the application is saved during the checkpointing process. One method to achieve this is to avoid rewriting portions of the process states that do not change between consecutive checkpoints.

In the proposed solution, the critical region was determined through a series of tests. It was found that the best range for the critical range would be 25% of Young's checkpointing interval. However, this conclusion is based on statistical analysis. A better method would be to use mathematical derivations to determine the best critical region for a given application. In this case, the researcher will need to study the occurrence of natural synchronisation points in parallel applications and then perform some mathematical research.

Another research area would be to develop an algorithm that will further improve the criteria to select the best options among the set of natural synchronisation points that may exist within a Critical Region. Ideally the Natural Synchronisation Point closest to the

Young's Checkpoint Interval is the best option. One method would be to buffer a checkpoint file temporarily, replacing it with the better option as we move along the execution line within a Critical Region. When we reach the end of the critical region, the checkpoint image in the buffer is confirmed as permanent checkpoint file.

References

[1] **M. Baker:** Ian Foster on Recent Changes in the Grid Community. IEEE Distributed Systems Online, February 2004.

[2] **K. Czajkowski et al:** From Open Grid Services Infrastructure to WSResource Framework: Refactoring & Evolution. March 2004. http://www-106.ibm.com/developerworks/library/ws-resource/ogsi_to_wsrf_1.0.pdf

[3] **E. Deelman et al:** Workflow Management in GriphyN. Grid Resource Management, J. Nabrzyski, J. Schopf, and J. Weglarz editors, Kluwer, 2003.

[4] **E. Deelman et al:** Mapping Abstract Complex Workflows onto Grid Environments.
Journal of Grid Computing, 2003.

[5] **V. Dialani et al:** Transparent Fault Tolerance for Web Services based Architectures. In the proceedings of the Eighth International Europar Conference (EURO-PAR'02). Padeborn, Germany, August 2002.

[6] **L. Ferreira et al:** Introduction to Grid Computing with Globus. 2003 http://www.redbooks.ibm.com/redbooks/pdfs/sg246895.pdf

[7] **I. Foster et al**: The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.

[8] **J.M Helary et al.:** Preventing Useless Checkpoints in Distributed Computations. SRDS 1997. Proceedings of the 16th Symposium on Reliable Distributed Systems, 1997.

[9] **A. Hoheisel:** Dynamic Workflows for Grid Applications. Cracow Grid Workshop 2003, Cracow, Poland, 2003.

[10] **W. Hoschek:** Introduction to Grids and Globus. March, 2000. http://wwwpdp.web.cern.ch/wwwpdp/te/globus/intro.html.

[11] **S. Hwang et al**: Grid workflow:A Flexible Failure Handling Framework for the Grid. 12th IEEE International Symposium on High Performance Distributed Computing, 2003.

[12] **S. Hwang and C. Kesselman:** A Flexible Framework for Fault Tolerance in the Grid. Journal of Grid Computing, September 2003.

[13] **B. Jacob et al:** Enabling Applications for Grid Computing with Globus. IBM Redbook, June 2003. http://www.redbooks.ibm.com/redbooks/pdfs/sg246936.pdf

[14] **J. Joseph et al:** Introduction to Grid Computing. http://www.informit.com/articles/article.asp?p=169508&seqNum=4

[15] **B. Sotomayor:** http://www.casa-sotomayor.net/gt3-tutorial/ multiplehtml/ pt01.html

[16] **Y. Robert et al.:** High Performance computing – HiPC 2006. 13th International conference, Bangalore, India, 2006

[17] **P. Townend and J. Xu:** Fault Tolerance within a Grid Environment. In Proceedings of U.K. e-Science 2nd All Hands Meeting, Simon J. Cox Eds., Nottingham Conference Centre, 2003.

[18] **S. Tuecke et al:** Open Grid Services Infrastructure (OGSI). Global Grid Forum Draft Recommendation, 2003.http://citeseer.ist.psu.edu/634383.html

[19] **A. N. Tuong:** Integrating Fault-Tolerance Techniques in Grid Applications, Ph.D. thesis, University of Virginia, 2000.

[20] WS-Resource Framework. Globus Alliance and IBM in conjunction with HP 2004. http://www.globus.org/wsrf.

[21] **D. Zou et al:** Fault-Tolerant Grid Architecture and Practice. Journal of Computer Science and Technology, 2003

[22] **S. Bogomolov, A. Bondarenko, A. Fyodorov:** Fault Tolerance Software Library for Support of Real-Time Embedded Systems.

[23] **J. S. Plank:** An overview of Checkpointing in Uniprocessor and Distributed Systems, focusing on Implementation and performance.

[24] **D. Pei et al.:** Design and Implementation of a Low-Overhead File Checkpointing Approach. The Fourth International Conference, High Performance Computing in the Asia-Pacific Region, 2000.

[25] **E.N. Elnozahy, J.S. Plank, W.K. Fuchs :** Checkpointing for Peta-Scale Systems: A Look into the Future of Practical Rollback-Recovery. Dependable and Secure Computing, IEEE Transactions, April 2004.

[26] **A. Mostefaoui and M. Raynal:** Efficient Message Logging for Uncoordinated Checkpointing Protocols. In Proceedings of EDCC, 1996.

[27] **L. M. Silva, J.G. Silva :** System-Level versus User-Defined Checkpointing. Reliable Distributed Systems proceedings. Seventeenth IEEE Symposium, October 1998.

[28] **R.Y Camargo et al.:** Checkpointing-based rollback recovery for parallel applications on the InteGrade grid middleware. MGC Proceedings of the 2nd workshop on Middleware for grid computing, 2004.

[29] **V. A. Ogale**: Try, try till you succeed: Multiple checkpointing and rollback in distributed systems. 2004

[30] **G. Zheng:** Checkpoint-based methods. https://charm.cs.uiuc.edu/papers/FTCCharm.www/node3.html

[31] **Open MPI**: Open Source High Performance Computing. Available online at: http://www.open-mpi.org/

[32] **G.Suri et al.:** Reduced overhead logging for rollback recovery in Distributed Shared Memory. In Proc. of the 25th Annual International Symp. on Fault-Tolerant Computing, 1995.

[33] **S. Mishra and D. Wang:** Choosing an Appropriate Checkpointing and Recovery Algorithm for Distributed Applcations. In 11th ISCA International Conference on Computers and their Applications, San Francisco, CA, March 1996.

[34] **A. Bouteiller et al.:** Coordinated checkpoint versus message log for fault tolerant MPI. IEEE International Conference on Cluster Computing, 2003.

[35] **J.S.Plank et al.:** Memory Exclusion: Optimising the performance of Checkpointing Systems. Technical Report UT-CS-96-335, University of Tennessee, August, 1996.

[36] **A. Goldchleger et al.:** Running Highly-Coupled Parallel Applications in a Computational Grid. Proceedings of the 22th Brazilian Symposium on Computer Networks, 2004.

[37] **G. Zheng, L.Shi and L.V.Kale:** FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI. Cluster Computing, IEEE International Conference, September 2004.

[38] **S. Krishnan:** An Architecture for Checkpointing and Migration of Distributed Components on the Grid. Indiana University Indianapolis, IN, USA. November 2004.

[39] **P. Kacsuk et al.:** P-GRADE: A Grid Programming Environment. Journal of Grid Computing, volume 1, issue 2, pages: 171 – 197, 2003.

[40] **J. Hursey et al.:** A Checkpoint and Restart Service Specification for Open MPI. Indiana University Computer Science tech report TR635. July 2006

[41] **J. Hursey et al.:** The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI. Parallel and Distributed Processing Symposium. March 2007

[42] **J. Duell et al.:** The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Berkeley Lab Technical Report. December 2002

[43] **J. W. Young:** A first order approximation to the optimum checkpoint interval. Communications of the ACM, volume 17, pages: 530 – 531, September 1974.

[44] **K. M. Chandy and L. Lamport:** Distributed snapshots - determining global states of distributed systems. ACM Trans. Comput. Syst.1985.

[45] **E. N. M. Elnozahy, et al.:** A survey of rollback-recovery protocols in message passing systems. ACM Computing Surveys. Volume 34 Issue 3, September 2002.

[46] **E. Garbriel, et al.:** Open MPI: goals, concept, and design of a next generation MPI implementation. In Proceedings, 11th European PVM/MPI Users' Group Meeting, 2004.

[47] **Message Passing Interface Forum**: MPI - A Message Passing Interface. In Proc. of Supercomputing '93, pages 878–883. IEEE Computer Society Press, November 1993.

[48] **S. Sankaran et al. :** The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. International Journal of High Performance Computing Applications, 2005.

[49] **S. Haider et al.:** Fault Tolerance in Distributed Paradigms. 2011 International Conference on Computer Communication and Management, Singapore, 2011.

[50] **M. Beck, J.S. Plank and G. Kingsley:** Compiler-Assisted Checkpointing. IEEE TCOS, 1995.

[51] Virtual organization (grid computing). Available online at: http://en.wikipedia.org/wiki/Virtual_organization_(grid_computing)

[52] **B. Monien and R. Feldmann:** Euro-Par 2002 Parallel Processing. 8th international Euro-Par conference Padeborn, Germany, August 2002.

[53] **G. K. Saha:** Fault tolerance in web services. ACM New York, USA, March 2006.

[54] Towards Open Grid Services Architecture. Available online at: http://www.globus.org/.

[55] CryoPID - A Process Freezer for Linux. http://cryopid.berlios.de/

[56] DMTCP: Distributed MultiThreaded CheckPointing. http://dmtcp.sourceforge.net/

[57] **J. Ansel, K. Arya and G. Cooperman:** DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. Parallel & Distributed Processing. IEEE International Symposium. IPDPS 2009.

[58] Order of Approximation: http://en.wikipedia.org/wiki/Orders_of_approximation

[59] Taylor Series Approximation: http://www.mathsci.sharif.edu

[60] Maclaurin Series: http://mathinsite.bmth.ac.uk/pdf/macseries_theory.pdf

[61] Parallel computing: http://en.wikipedia.org/wiki/Parallel_computing

Bibliography

J. Duell, P. Hargrove, and E. Roman: Requirements for Linux Checkpoint/Restart. Berkeley Lab Technical Report, May 2002

J. Duell, P. Hargrove, and E. Roman: The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart. Berkeley Lab Technical Report, December 2002

E. Roman: A Survey of Checkpoint/Restart Implementations. Berkeley Lab Technical Report, July 2002

S. Sankaran et al: The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In LACSI Symposium, October 2003.

P. H. Hargrove and J.C. Duell: Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters In Proceedings of SciDAC 2006: June 2006.

Y. Bertot, B. Grégoire and X. Leroy: A Structured Approach to Proving Compiler Optimizations Based on Dataflow Analysis. In Proc. TYPES, 2004, pp.66-81.

E. Roman: A Survey of Checkpoint/Restart Implementations Lawrence Berkeley National Laboratory Technical Report, 2003

Y.Ling, J.Mi and X.Lin: A Variational Calculus Approach to Optimal Checkpoint Placement. IEEE Trans. Computers 50(7): 699-708 (2001)

J. Christmansson, M. Hiller and M. Rimén: An Experimental Comparison of Fault and Error Injection. Proceedings of the 9th International Symposium on Software Reliability Engineering (ISSRE-9), pp. 396-378, 1998

A. Ziv and J. Bruck: An on-line algorithm for checkpoint placement. IEEE Trans. Computers, vol. 46, pp. 976-985, September 1997.

M. Russinovich, Z. Segall and D. P. Siewiorek: Application Transparent Fault Management in Fault Tolerant Match. FTCS 1993: 10-19

Y.Tamir and T.M.Frazier: Application-Transparent process-level error recovery for multicomputers. In Proceedings of the Hawaii International Conference on System Sciences, pp. 296-305, January 1989

G. Bronevetsky et al: Application-level checkpointing for shared memory programs. In

ASPLOS(2004) 235-247

R.Y.D. Camargo, A. Goldchleger, F. Kon, and A. Goldman: Checkpointing BSP parallel applications on the InteGrade Grid middleware. Presented at Concurrency and Computation: Practice and Experience, 2006, pp.567-579.

M. Beck, J. S. Plank and G.Kingsley: Compiler-assisted checkpointing .Technical report UT-CS-94-269, University of Tenessee, December, 1994.

F. Karablieh, R.A. Bazzi, and M. Hicks: Compiler-Assisted Heterogeneous Checkpointing. In Proc. SRDS, 2001, pp.56-56.

J. Long, W.K. Fuchs, and J.A. Abraham: Compiler-Assisted Static Checkpoint Insertion. In Proc. FTCS, 1992, pp.58-65.

C. C. Li and W. K. Fuchs: Compiler-Assisted Full Checkpointing. IEEE Transactions on Software Engineering, Submitted 1991.

C-C. J. Li and W. K. Fuchs: CATCH - Compiler-Assisted Techniques for Checkpointing. 20th International Symposium on Fault Tolerant Computing, 1990.

A. N. Norman, S.-E. Choi, and C. Lin: Compiler-generated staggered checkpointing. In Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems (LCR), pages 1–8, New York, NY,USA, 2004.

J. Plank, M. Beck and G. Kingsley: Compiler-assisted memory exclusion for fast checkpointing. IEEE Technical Committee on Operating Systems and Application Environments, Special Issue on Fault-Tolerance, 1995.

J. Hong, S. Kim and Y. Cho: Cost Analysis of Optimistic Recovery Model for Forked Checkpointing. IEICE Transactions on Information and Systems, Vol. E86-D, No. 9, pp. 1534–1541, Septempber 2003

P. Kacsuk, T. Kiss and G. Sipos: Solving the Grid Interoperability Problem by P-GRADE Portal at Workflow Level, Future Generation Computing Systems: International Journal of Grid Computing: Theory, Methods and Applications, Volume 24, July 2008.

D.B. Johnson: Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs. In Proceedings of the 12th Symposium on Reliable Distributed Systems (SRDS 1993), pp. 86-95, IEEE Computer Society, Princeton, NJ, October 1993.

D. Thain and M. Livny: Error Scope on a Computational Grid: Theory and Practice. In Proceedings HPDC, 2002.

J. Leon, A. L. Fisher, and P. Steenkiste: Fail-safe PVM: a portable package for distributed programming with transparent recovery. Carnegie Mellon University, CMU-CS-93-124, February 1993.

J.C. Smolens et al.: Fingerprinting: Bounding Soft-Error-Detection Latency and Bandwidth. *IEEE Micro*, Vol. 24, No. 6, November, 2004

G. Suri, B. Jannsens and W.K. Fuchs: Reduced Overhead Logging for Rollback Recovery in Distributed Shared Memory. Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995

F. Warg and P. Stenström, "Reducing misspeculation overhead for module-level speculative execution", in Proc. Conf. Computing Frontiers, 2005, pp.289-298.

P.E. Chung et al.: Winckp: A Transparent Checkpointing and Rollback Recovery Tool for Windows NT Applications. In Proc. FTCS, 1999.

O. Laadan, D.B. Phung and J. Nieh: Transparent Checkpoint-Restart of Distributed Applications on Commodity Clusters. In Proc. CLUSTER, 2005.

M. Rieker, J. Ansel and G. Cooperman: Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux. In Proc. PDPTA, 2006.

L.M. Silva, J.G. Silva, and S. Chapple: Portable Transparent Checkpointing for Distributed Shared Memory. In Proc. HPDC, 1996.

D.F. Bacon: Transparent Recovery in Distributed Systems. Presented at Operating Systems Review, 1991, pp.91-94.

T. M. Frazier and Y. Tamir: Application-Transparent Error-Recovery Techniques for Multicomputers. Proceedings of the 4th Conf on Hvpercubes, Concurrent Computers and Applications . Monterey, CA, Mar. 1989.

E.N. Elnozahy and W. Zwaenepoel: Manetho-Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. Presented at IEEE Trans. Computers, 1992.

K. H. Kim: Programmer-Transparent Coordination of Recovering Concurrent Processes: Philosophy and Rules for Efficient Implementation. In Proceedings of IEEE Trans. Software Eng. 1988.

Appendices

.1 Publications

K. Sajadah, G. Terstyanszky, S.Winter and P. Kacsuk: Checkpointing of parallel applications in a grid environment. In: Kacsuk, Peter K. and Lovas, Robert and Nemeth, Zsolt, (eds.) Distributed and parallel systems: in focus: desktop grid computing. Springer, Boston, MA, pp. 179-187. ISBN 9780387794471, 2008

K. Sajadah, G. Terstyanszky, S.Winter and P. Kacsuk: A checkpointing mechanism for the Grid environment. In: Proceedings of the UK e-Science All Hands Meeting 2008, Edinburgh, UK, 8th - 11th September 2008. National e-Science Centre, Edinburgh, 2008

.2 Sample Checkpoint File

The checkpoint file contains the necessary information necessary to restart an application fro a given point after failure. This section analyses the content of a checkpoint file.
The “ompi_global_snapshot_1616.ckpt” file for example contains directories: 0, 1, 2, 3 and 4 and files “global_snapshot_meta.data” and “restart-appfile”.
Each of these directories contains the checkpoint file of each process obtained using the BLCR uniprocess checkpointing solution.
For example folder “0” contains the following directories:

```
opal_snapshot_0.ckpt
opal_snapshot_1.ckpt
opal_snapshot_2.ckpt
opal_snapshot_3.ckpt
```

“opal_snapshot_0.ckpt” for example contains the checkpoint file for process 0. The directory contains the following files:

```
ompi_blcr_context.1629
snapshot_meta.data
```

“ompi_blcr_context.1629” is the actual checkpoint file created by the BLCR checkpointing mechanism.
“snapshot_meta.data” gives information about the checkpointing file. For example, it will give information about the process ID, the tools used to perform checkpointing and the name of the checkpointing file.

The “global_snapshot_meta.data” file is used to put all the local checkpoint files into a global checkpoint file. It contains a sequence number for each global checkpoint performed.

The following snapshot of the file is obtained when the first checkpoint is performed:

```
#############################################################
# Seq: 0
# Timestamp: Sun Jul 18 23:56:18 2010
# Process: 1574699009.0
# OPAL CRS Component: blcr
# Snapshot Reference: opal_snapshot_0.ckpt
# Snapshot Location: /home/raj/ompi_global_snapshot_1616.ckpt/0
# Process: 1574699009.1
# OPAL CRS Component: blcr
# Snapshot Reference: opal_snapshot_1.ckpt
# Snapshot Location: /home/raj/ompi_global_snapshot_1616.ckpt/0
# Process: 1574699009.2
# OPAL CRS Component: blcr
# Snapshot Reference: opal_snapshot_2.ckpt
# Snapshot Location: /home/raj/ompi_global_snapshot_1616.ckpt/0
# Process: 1574699009.3
# OPAL CRS Component: blcr
# Snapshot Reference: opal_snapshot_3.ckpt
# Snapshot Location: /home/raj/ompi_global_snapshot_1616.ckpt/0
# Timestamp: Sun Jul 18 23:56:18 2010
# Finished Seq: 0
#############################################################
```

The “Seq:0” implies it is the first checkpoint taken. Snapshot Reference references a process number. For example “Snapshot Reference: opal_snapshot_0.ckpt” references the process 0.

Snapshot Location gives the location of a given checkpoint file. For example: “Snapshot Location: /home/raj/ompi_global_snapshot_1616.ckpt/0” indicates where “opal_snapshot_0.ckpt” is found.

Using these files, Openmpi is able to create a global checkpointing file.

The “restart-appfile” file is normally formed if the application has been restarted. The content of the file depends on the restart command used to restart the application.

For example, if we want to restart and application from the second checkpoint file created, we will use the following command:

```
ompi-restart -s 1 ompi_global_snapshot_1616.ckpt
```

(N.B:- if you are running on the cluster, use: ompi-restart -s 1 -mca btl ^openib -mca snapc_base_global_snapshot_dir /tmp ompi_global_snapshot_12941.ckpt).

This will create the "restart-appfile" file and the content of the file will be as follows:

```
############################
#
# Old Process Name: 1574699009.0
#
-np 1 -am ft-enable-cr opal-restart -mca crs_base_snapshot_dir
/home/raj/ompi_global_snapshot_1616.ckpt/1 opal_snapshot_0.ckpt
#
# Old Process Name: 1574699009.1
#
-np 1 -am ft-enable-cr opal-restart -mca crs_base_snapshot_dir
/home/raj/ompi_global_snapshot_1616.ckpt/1 opal_snapshot_1.ckpt
#
# Old Process Name: 1574699009.2
#
-np 1 -am ft-enable-cr opal-restart -mca crs_base_snapshot_dir
/home/raj/ompi_global_snapshot_1616.ckpt/1 opal_snapshot_2.ckpt
#
# Old Process Name: 1574699009.3
#
-np 1 -am ft-enable-cr opal-restart -mca crs_base_snapshot_dir
/home/raj/ompi_global_snapshot_1616.ckpt/1 opal_snapshot_3.ckpt
################################
```

This file describes the location of the local checkpoints files of the four processes created when the second checkpoint was triggered. The "Old Process Name" is the name given to each process in the will "global_snapshot_meta.data" file.