2012-01-01

Fault Tolerance: Validating A Mathematical Model Via A Case Study Of RAxML, An HPC Community Code

Bidisha Chakraborty
University of Texas at El Paso, bchakraborty@miners.utep.edu

FAULT TOLERANCE: VALIDATING A MATHEMATICAL MODEL VIA A CASE STUDY OF RAxML, AN HPC COMMUNITY CODE

BIDISHA CHAKRABORTY

Department of Computer Science

APPROVED:

Patricia J. Teller, Ph.D., Chair

Sarala Arunagiri, Ph.D., Co-Chair

Michael P. McGarry, Ph.D.

Benjamin C. Flores, Ph.D.
Interim Dean of the Graduate School

to my

Baba and Maa

and Aritra

with love

FAULT TOLERANCE: VALIDATING A MATHEMATICAL MODEL VIA A CASE STUDY OF RAxML, AN HPC COMMUNITY CODE

by

BIDISHA CHAKRABORTY

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2012

Acknowledgements

I would like to express my deep-felt gratitude to my advisor, Dr. Patricia J. Teller of the Computer Science Department at The University of Texas at El Paso, for her advice, encouragement, enduring patience and constant support. She was never ceasing in her belief in me (though I was often doubting in my own abilities), always providing clear explanations when I was lost, constantly driving me with energy.

Another person who has always been motivating me regarding my work was Dr. Sarala Arunagiri of Computer Science Department at University of Texas at El Paso. She always motivated me to work to excel. Her advice and support has helped me understand and implement new ideas.

I also wish to thank the Dr. Michael P. McGarry of the Electrical Engineering Department, at The University of Texas at El Paso. His suggestions, comments and additional guidance were invaluable to the completion of this work.

Additionally, I want to thank The University of Texas at El Paso Computer Science Department professors and staff for all their hard work and dedication, providing me the means to complete my degree and prepare for a career as a computer scientist.

Also I would like to thank Michael Harney, my fellow researcher, who has always helped me with new ideas.

Last but not the lest I would lie to thank my husband Aritra for his love and support.

NOTE: This thesis was submitted to my Supervising Committee on the Apr 15, 2012.

Abstract

Researchers have mentioned that the three most difficult and growing problems in the future of high-performance computing will be avoiding, coping and recovering from failures. As the scale of computing increases, the Mean Time to Failure (MTTF) of the entire system decreases and, therefore, system resilience and fault tolerance techniques become mandatory. One of the most commonly used fault tolerance schemes is checkpoint/restart, however, it has been predicted that the current checkpoint/restart approach is not scalable. Thus, current research seeks to find scalable fault tolerance techniques as well as to extend the scalability of checkpoint/restart.

The periodicity of the checkpointing operation, otherwise known as the checkpoint interval, plays an important role in application execution time and I/O performance. It can have a significant impact on execution time and the number of checkpoint I/O operations performed by the application. The frequency of checkpoint I/O operations performed by the application, along with its productive I/O, determine the demand made by the application on the I/O bandwidth of a massively parallel processing (MPP) system. There are analytical models for finding the optimal checkpoint interval that minimizes wall-clock execution time and the optimal checkpoint interval that minimizes the number of checkpoint I/O operations generated.

This thesis presents a study that quantitatively measures the effect of the checkpoint interval on workload execution time and the number of checkpoint I/O operations generated. The study is based on the execution behavior of RAxML 7.2.6, a popular community code, and RAxML-Light 1.0.6 on an HPC system as well as simulations of workloads executed on an HPC system. Parameter values for the HPC runs and the simulations are the product of analysis of historic failure data of 22 systems made available by the Computer Failure Data Repository (CFDR). This analysis also is presented in the thesis.

Our research showed that increasing the checkpoint interval to a value above the optimal

checkpoint interval with respect to the execution time results in a significant decrease in the number of checkpoint I/O operations with a marginal increase in execution time. This shows that the associated analytical model holds good for the cases studied.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

"Going to the Exascale" is the latest challenging endeavor towards which a substantial body of research in high performance computing is directed. Exascale computing means millions of processors working together in tandem. As described in a recent report [58], Exascale computing will require a radical change in every aspect of computer organization, architecture and design. It also will require transformations in application programming models, compilers, algorithms and application codes. In addition to challenges in hardware, power consumption, and memory organization/usage, one of the biggest challenges in this path towards Exascale is predicted to be in the area of system resilience [28]. Researchers have mentioned that the three most difficult and growing problems in the future will be avoiding, coping and recovering from failures. As the scale of computing increases, the Mean Time to Failure (MTTF) of the entire system decreases and, therefore, system resilience and fault tolerance techniques become mandatory.

Currently, one of the commonly used fault tolerance schemes is checkpoint/restart [28], [58], [24]. In the checkpoint/restart scheme, the global state of an application is periodically written as a checkpoint file to persistent storage, like disks, to allow restoration of state in the case of a failure. In the event of a failure, the checkpoint/restart scheme enables the application to restore its state to one prior to the failure, using information from the latest stored checkpoint file. Checkpoint/restart can be either application driven or system driven. In application-driven checkpoint/restart, the application decides when to perform the checkpoint write, while in system-driven checkpoint/restart, the system periodically stores the state of all active applications to disks. Another classification of the checkpoint/restart strategy is coordinated and non-coordinated checkpoint using message

logging [24]. In coordinated checkpointing, all processes performing a checkpoint coordinate to store a stable global state to disk storage. Since there is no global clock in MPP systems, capturing the state of different processes running on different nodes is difficult. Thus, processes communicate and synchronize among themselves to store their local states and the state of their interactions within the network. In case of a failure, all processes need to be restarted. In a non-coordinated checkpoint strategy using message logging, the individual processes store their network states as messages to the server. They perform a checkpoint individually. It is assumed that process execution is piecewise deterministic [57], i.e., it is governed by its message reception [36]. During a failure, only the failed processes are restarted. Thus, a failed process is restored to its prior state and executes the same computation as in its previous execution, which is based on messages stored on the stable storage.

During traditional checkpointing, the entire state of the application is stored to a persistent storage, e.g., disk storage, periodically (periodic checkpoint system). In MPP systems, the overhead of an I/O operation is enormous. The I/O disks are shared among multiple nodes of the cluster and, therefore, an I/O operation contributes to network contention as well. This overhead ultimately affects the execution time of the application and all active concurrently-executing applications. Thus, a reduction in the number of I/O operations can be expected to have a positive impact on the overall performance of a high-end computing (HEC) system.

It is predicted that traditional checkpoint/restart approach is not scalable[15]. One of the main reasons for non-scalability of checkpoint/restart techniques for large-scale systems is the sheer volume of bursty checkpoint data that is required to be handled by the I/O system. Accordingly, there is research being conducted that is related to building fault resilient algorithms. But designing such approaches is a challenge due to the fact that such fault tolerance is tightly coupled with the system itself. However, fault tolerance mechanisms should be portable and not dependent on the systems on which they will run. Some studies show that coordinated checkpointing is less scalable than the non-

coordinated message logging checkpoint/restart strategy. There also is research related to reducing checkpoint data size by performing incremental checkpoints [9]. This research is not directed towards reducing checkpoint data sizes, but it can be used in conjunction with techniques for reducing the size of checkpoint data, potentially resulting in further improvement in checkpoint/restart efficiency by minimizing the stress on the I/O system. Also, recent studies are geared towards using in-memory checkpoint [67], checkpointing to solid state devices (SSDs) [47]. Although such research can lead to a better fault tolerance scheme, this goal can be facilitated by the study of historic failure data, which can provide information about failures, their types and origins.

The remainder of this chapter presents the motivation for doing the research discussed in this thesis (Section 1.1), the common terminology used in this thesis (Section 1.2), the problem description and a brief description of our solution (Section 1.3), a high-level description of our experimental methodology (Section 1.4), and a list of the contributions of this thesis as well as the organization of the remainder of the thesis (Section 1.5).

1.1 Motivation

Parallel applications can benefit from high-performance computational capabilities of Petascale systems. However, I/O performance is the greatest slowdown factor of Petascale applications and, therefore, the main performance bottleneck arises from I/O latency [18]. The rate of growth of disk-drive performance, both in terms of the number of I/O operations per second and sustained bandwidth, is smaller than the rate of growth of the performance of other memory components in computing systems [59]. As shown in Figure 1.1, disk access is slower than SRAM and DRAM access by several orders of magnitude. As a result, a reduction in the number of I/O operations is expected to have a positive impact on the overall performance of MPP systems.

For HPC applications I/O operations can be classified into two broad categories: productive I/O and defensive I/O. I/O operations that are inherent to the application are

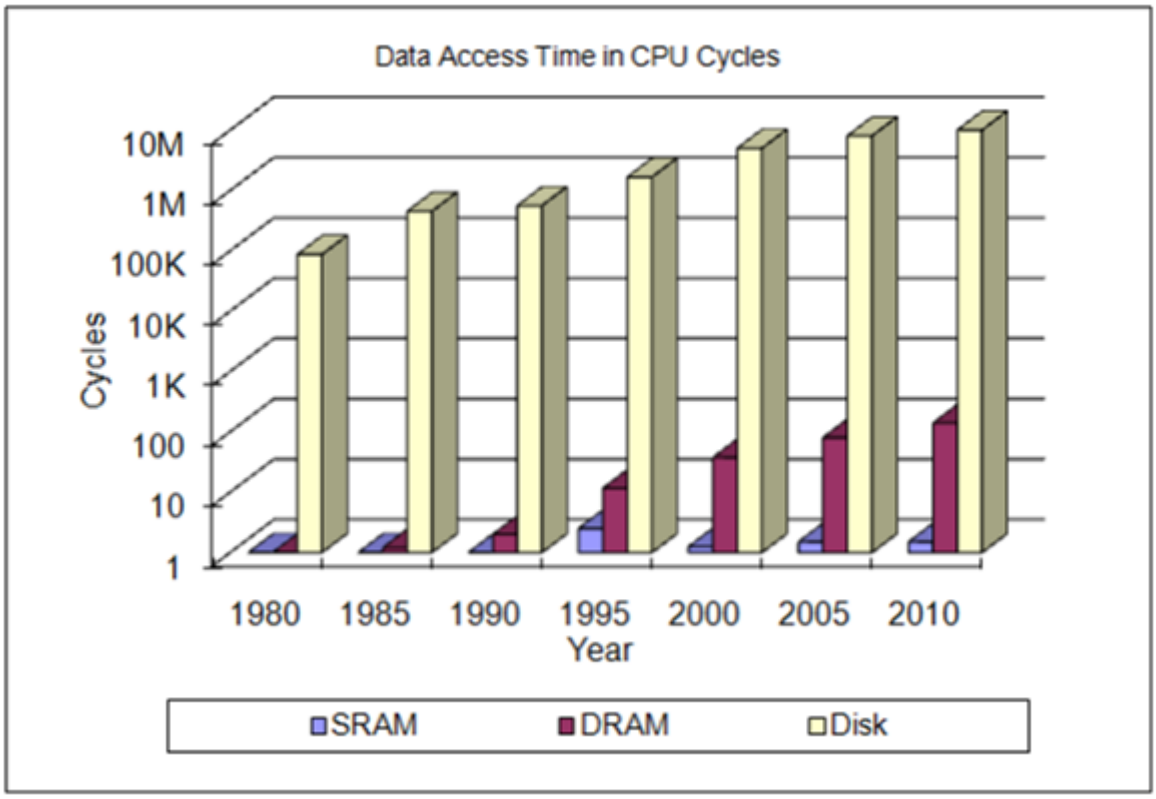


Figure 1.1: Memory Access Times in CPU Cycles

called productive I/O operations. Productive I/O operations are mandatory and are used during various processes like visualization, rendering, etc. Defensive I/O operations are optional and generally related to fault tolerance methods like checkpoint I/O.

For large applications, it is observed that about 75% of the overall I/O is defensive [3]. Studies show that when HPC applications run with higher process counts, i.e., on multiple nodes, their checkpoint frequencies and checkpoint file sizes increase [39]. Another feature of checkpoint I/O is that it occurs in bursts [23], [35], [45], [42]. Thus, frequent checkpoint writes will affect the network contention of the system. Accordingly, a systematic effort to reduce the number of checkpoint IO operations is required. Although, there is an ongoing debate about whether or not the checkpoint/restart technique will be used for fault tolerance in Exascale systems, until now there has been no stable fault tolerance

technique that can replace checkpoint/restart completely. This study focuses on reducing the number of checkpoint I/O operations in tandem with other methods that can increase the scalability of checkpoint/restart.

1.2 Terminology

While discussing the checkpoint/restart strategy, which is also referred to as the rollback-/recovery strategy, certain terms are commonly used; they are presented below.

- **Checkpoint latency:** The amount of time required to write the checkpoint file to persistent storage, like disk storage.
- **Checkpoint interval:** The application execution time between two consecutive checkpoint operations.
- **Checkpoint overhead:** The increase in the execution time of an application due to checkpointing.
- **Solution time (of an application):** The time taken by the application to complete execution without any failures or fault tolerance measures.

Figure 1.2 gives a pictorial representation of these terms.

1.3 Problem Definition

The periodicity of the checkpointing operation, otherwise known as the checkpoint interval, plays an important role in the execution time and I/O performance of the application. Consider an application having a total solution time of x hours (without any defensive I/O operations). The total time taken to complete application execution in a failure-free environment with no other concurrently-executing applications primarily depends on two factors:

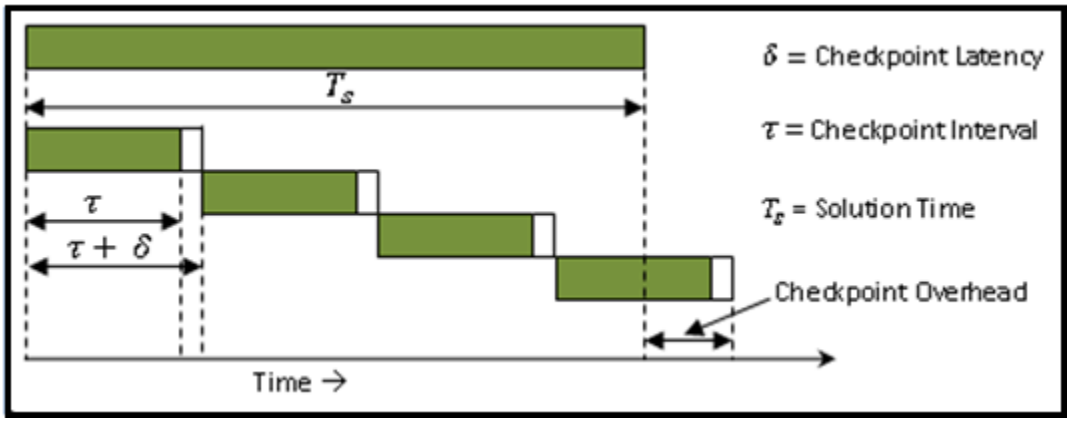


Figure 1.2: Checkpoint Latency, Interval and Overhead as well as Solution Time

- I/O latency to write a checkpoint file and
- the number of checkpoint writes.

A recent report [15] states that for Top500 machines the average time to perform a checkpoint of the entire system (alias checkpoint latency) varies from 20 to 30 minutes. The checkpoint latency is dependent on the size of the checkpoint file, system characteristics and the other applications running in parallel and contending for resources such as the network and I/O subsystem. The checkpoint interval is the only parameter in an application-driven periodic-checkpointing system that can be modified by an application developer to yield different results in terms of the number of I/O operations and total wall clock execution time. However, to do this, the application developer needs to understand the relationship of the application execution time and the number of I/O operations to the checkpoint interval this is a challenging task.

Research Objective: To enhance the understanding of the behavior of real HPC applications in terms of their execution times and the number of I/O operations they generate as a function of their checkpoint intervals, we have used information from existing analytical models and historic failure data. The two analytical models considered in our study are briefly presented in the following subsections.

1.3.1 Analytical Model for Wall Clock Execution Time

John Daly [19],[20],[21] presented a model to estimate the optimal checkpoint interval for systems that follow a Poisson failure distribution. Daly showed that the execution time of an application can be defined as:

$$T = Me^{\frac{R}{M}}(e^{\frac{(\tau+\delta)}{M}} - 1)\frac{T_s}{\tau}$$

His hypothesis showed that the optimal checkpoint interval, $\widetilde{\tau_{opt}}$, which would result in minimum execution time, can be defined as:

$$\widetilde{\tau_{opt}} = \begin{cases} \sqrt{2\delta M}\left[1 + \frac{1}{3}\left(\frac{\delta}{2M}\right)^{\frac{1}{2}} + \frac{1}{9}\left(\frac{\delta}{2M}\right)\right] - \delta & \text{for } \delta < 2M \quad (1.1a) \\ M & \text{for } \delta \geq 2M \quad (1.1b) \end{cases}$$

where,

T_s = Solution time (time taken by an application to complete computation without any failures and fault tolerant measures),

τ = Checkpoint interval (time elapsed between each checkpoint file write),

δ = Checkpoint latency (time taken to write a checkpoint file to persistent storage),

R = Restart time (time taken to recover from any failure), and

M = Mean Time to Interrupt (MTTI) an application (MTTI refers to the mean time between application interruptions. The mean time between failures or the time between occurence of two failures in a system, is defined as a sum of the mean time to repair (MTTR) and mean time to interrupt (MTTI). The MTTI is used in to replace mean time to failures (MTTF) for practical cases [15]).

Accordingly, he showed that the optimal checkpoint interval does not depend on the applications solution time but, rather, on the latency of the checkpoint operation and the MTTI of the system. The latency to write a checkpoint file depends on the amount

of data written during a checkpoint (size of the checkpoint file) and on the state of the network. For applications where the size of the checkpoint file changes over time, the optimal checkpoint interval will change dynamically. Also for applications with similar amounts of checkpoint data, the checkpoint interval will be similar. Mathematically this optimal checkpoint interval will provide minimum execution time.

1.3.2 Analytical Model for Number of I/O Operations

S. Arunagiri, et al presented an analytical model, based on Daly's execution time model, for the number of defensive I/O operations [12]. In the paper, the authors show that, similar to Daly's Execution time model, the number of I/O operations is also a function of checkpoint interval, checkpoint latency, MTTI, solution time and restart time.

$$N_{I/O} = \frac{T_s}{\tau}\left[1 + e^{\frac{R}{M}}\left(e^{\frac{\delta+\tau}{M}} - 1\right)\right]$$

The checkpoint interval that leads to the minimum number of defensive I/O operations, the optimal checkpoint interval with respect to the number of I/O operations is:

$$\tau_{I/O} = M\left[1 + ProductLog\left(-e^{-\frac{\delta+M}{M}} + e^{-\frac{R+\delta+M}{M}}\right)\right]$$

where,

T_s = solution time, τ = checkpoint interval, δ = checkpoint latency, R = restart time, and M = mean time to interrupt (MTTI) an application.

It has also been proven that $\tau_{I/O} >> \tau_{opt}$.

The authors present a simulation-based study to validate their model. They present the plot depicted in Figure 1.3 to demonstrate that increasing the checkpoint interval has a greater impact on reducing the number of I/O operations than on reducing execution time. In this casestudy, they considered an application with Solution Time (T_s) = 500 hours, MTTI (M) = 24 hours, Checkpoint latency (τ) = 5 minutes and Restart Time (R) = 10 minutes. As shown in the figure, the blue line, shows the change in the execution time with increasing checkpoint interval and the green line represents, the decrease in number of I/O

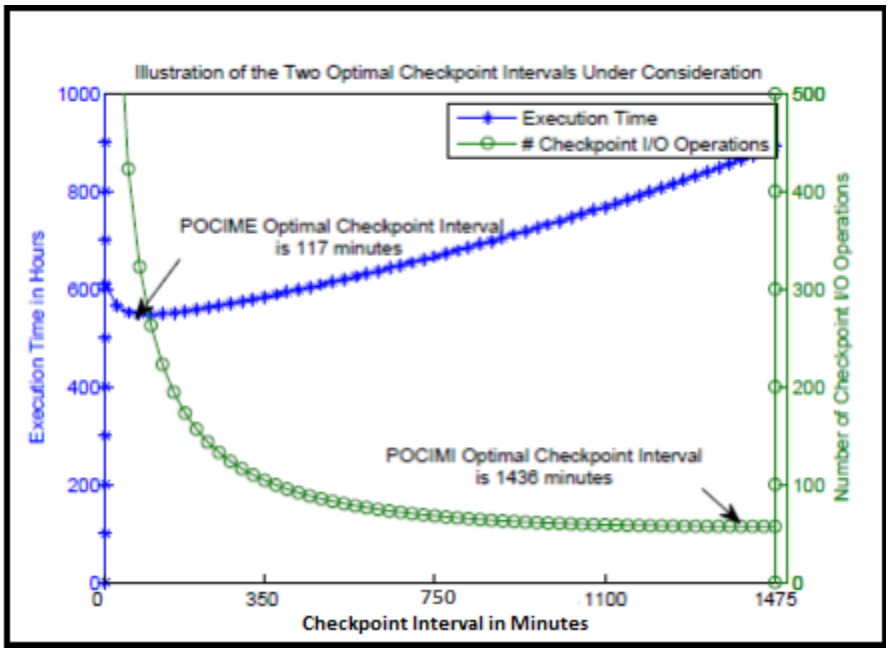


Figure 1.3: Optimal Checkpoint Interval given by Model of Execution Time vs. that given by Model of Number of I/O Operations.

operations with increasing checkpoint interval. From the curve it is evident, that the rate of decrease of number of I/O operations is higher than the rate of increase of execution time with respect to the increase in checkpoint interval. Thus, a larger checkpoint interval has a potential for a drastic decrease in the number of I/O operations accompanied by some increase in execution time. This is the main motivation behin our research which is directed towards studying the performance and significance of this model for real-life applications. The main objectives of our research are:

- Study the variation of the execution time and the number of I/O operations as a function of the checkpoint interval.
- Investigate patterns in historic failure data to provide information for realistic values of expected node and system MTTI and for scheduling of checkpoints.
- Using historic failure data, conduct a simulation-based study of a selected HPC ap-

plication to determine the percentage error of the estimates of execution time and number of I/O operations made by the two analytical models.

A secondary objective of this research is geared towards trying to predict the MTTI of the system. In this regard, we plan to use the MTTI computed based on n years of historic failure data, and determine the percentage error of the estimates of execution time and number of I/O operations made by the two analytical models using that MTTI in their computation.

1.4 High-level Description of Methodology

To study the variation of execution time and number of I/O operations as a function of the checkpoint interval in a real HPC application, based on consultation with experts at Texas Advanced Computing Center (TACC) [6], we selected RAxML, a popular community code. RAxML (**R**andomized **Ax**elerated **M**aximum **L**ikelihood) [53] is an important HPC application used for the phylogenetic study of evolutionary trees. This code has been used as a part of the NSF iPlant Tree of Life (iPTOL) Project [7]. In 2009, RAxML was reported to have around 6,000 source code downloads and over 40,000 job submissions to the two web-servers. It uses DMTCP [11] for checkpoint support and only the PThreads version of RAxML supports checkpointing. Since RAxML could not scale to more than one node, we moved to the stripped-down version of RAxML called RAxML-Light v.1.0.6. RAxML-Light can run on multiple nodes because it has support for MPI and is capable of performing checkpointing. Through a computational resource time allocation award, the application was executed on Ranger [8] at TACC with failures simulated based on the assumption of Poisson component failure. For all experiments, checkpoint/restart data was gathered and analyzed.

To investigate patterns in historic failure data, we analyzed failure records from 22 clusters at Los Alamos National Laboratory (LANL) collected over a duration of 10 years this data is available in the Computer Failure Data Repository (CFDR) [4]. Finally, the

checkpoint/restart parameters collected from running RAxML on TACC and historic failure data were used in a simulation study to determine the percentage error of the two analytical models.

1.5 Contributions of Thesis

The contributions of this thesis are:

- Using historic failure data, analysis of failure distribution and computation of expected node and system MTTI.
- Study of the variation of execution time and number of I/O operations as a function of checkpoint interval. We also investigated the validity of the analytical models related to periodic checkpoint/restart, i.e., the models for execution time [21] and number of I/O operations [12].

This investigation was carried out in two stages:

1. Execution of RAxML and RAxML-Light on a real MPP system (Ranger) with failures simulated assuming Poisson component failure, leading to a simulated exponential failure distribution.
   (a) To perform this study the RAxML and RAxML-Light applications were modified to incorporate logging functions to log the start time, end time and amount of data transfer, in bytes, for each checkpoint I/O operation.
   (b) The codes were modified to read from an external failure file and simulate failure and restart sequences after particular durations of time.
   (c) The RAxML-Light code was further modified to perform periodic checkpointing at regular intervals of time.

2. Using latency parameters from experiments on Ranger and MTTI estimates from analysis of the LANL failure data, a simulation study that is used to validate the models.

The main focus of this thesis is reducing the number of defensive I/O operations of an HPC application by modifying the checkpoint interval. The thesis is organized as follows: Chapter 2 presents background and related work; Chapter 3 describes the experimental methodology used to conduct this research; Chapter 4 presents and discusses the experimental results; and Chapter 5 states the conclusions and discusses future work.

Chapter 2

Related Work

2.1 Checkpoint/Restart Strategy

The research related to checkpoint/restart technique for fault tolerance is vast and there is extensive literature available in this domain. The checkpoint/restart strategy can be classified into three categories based on the style of operation used to implement the technique, namely independent checkpointing, coordinated checkpointing, and communication-induced checkpointing [24]. In independent checkpointing [13], the processes perform checkpointing individually without communicating with other processes. The main issue with this process is susceptibility to the domino effect. The domino effect is the process of repeatedly rolling-back process states (among multiple processes) until a consistent global state is realized; one rollback may cause multiple rollbacks to realize a global state. Although independent checkpointing has less synchronization overhead, coordinated checkpointing is the more commonly used checkpoint/restart strategy. In coordinated checkpointing [17], the parallel processes communicate with each other to save a system-wide consistent state. In the event of a failure, after restart, this information, which is stored in persistent storage, can be used to restore the state of the application without loss of consistency. The main concern with coordinated checkpointing is its scalability due to synchronization overhead. To address this, another checkpoint strategy has been proposed in the literature, which is called communication-induced checkpointing [40], [14], [60]. It is both asynchronous and independent, therefore, reducing communication overhead.

To achieve transparent checkpointing in computing clusters, coordinated checkpointing is the most commonly implemented checkpoint strategy. As mentioned above, in this

strategy all coordinating processors are synchronized to ensure consistent global state before checkpointing. Chandy, et al. present an algorithm that can be used to find the global state of an application running on a distributed system [17]. The main concept for attaining a global state is that each process must save its local state and communicate this action to the master process via a message. However, the main drawback with coordinated checkpointing is scalability. Pattabiraman, et al., show an efficient way to model coordinated checkpointing for large-scale systems [48]. They use the metric of useful work, i.e., computation that contributes to final job completion, for their performance evaluation. Thus, computation lost due to a failure is not useful work. They also suggest that for modeling useful work Stochastic Activity Networks (SANs) give better results than Markov models. This model considers all conditions and overheads related to coordinated checkpointing along with failure during checkpoint/restart.

One important question related to periodic checkpointing is how often a checkpoint write operation should be performed. There is a body of research that addresses this question and it presents models for determining the optimal checkpoint interval. One of the initial models was proposed by Young [65]. In this model, the optimal checkpoint interval is defined as a function of checkpoint latency/overhead (time to write a checkpoint file to persistent storage) and Mean Time to Interrupt (MTTI) for the system. However, this model assumes that no failure can occur during a checkpoint/restart operation as it assumes the Mean Time between Failures (MTBF) of the system is large compared to the checkpoint/restart time. However, this is not always true. Thus, in order to relax this assumption, Daly [21] proposed a higher order estimate of the optimal checkpoint interval based on [65]. This model considers the situation of a system failure occurring during a checkpoint/restart operation. The main focus of this work is on finding an optimum checkpoint/restart strategy that will minimize wall-clock execution time for applications running on systems exhibiting the Poisson single-component failure distribution. Initially, the paper presents a first-order model assumes that the optimal checkpoint interval may depend on the restart time. However, the higher-order model, which was developed based

on the first-order model, removes this dependency. It also presents a perturbation solution of the higher-order function to provide an accurate approximation to the optimal checkpoint interval. Note, however, that it does not model the overhead of coordinated checkpointing.

Subramaniyan, et al. illustrate another analytical model to determine optimal checkpoint frequency in terms of the MTBF of the system, the amount of memory checkpointed, the sustainable I/O bandwidth of the system, and the frequency of checkpointing [59]. This model studies a way to optimize I/O related to checkpoints.

Another important question related to this field of inquiry is whether checkpoints should be periodic or not. In this regard, Ling, et al. proposed another optimal checkpoint scheduling model based on the calculus of variation techniques [37]. The authors deduced a relationship between the optimal checkpoint interval and the failure distribution of the system. In their work they proposed that a periodic checkpoint strategy is optimal if the system failure trends follow a Poisson/exponential process and a non-periodic checkpoint scheme is optimal for systems not conforming to the Poisson failure distribution.

Jones, et al. present a study of sub-optimal checkpoint intervals in [33] based on [21]. They show that underestimating the optimal checkpoint interval has more adverse effects on application efficiency than overestimating it.

Various studies have shown that a periodic coordinated checkpoint/restart strategy will not scale effectively for Petascale and Exascale computing [15], [28]. There have been various kinds of efforts to find a more suitable and scalable fault tolerance technique, such as fault-tolerant algorithms [63], [27] and replication [29], self healing [26] and self stabilization [51]. Also, there are studies on enhancing checkpoint I/O performance by replacing disk-based checkpoints with in-memory checkpoints [67], staging I/O on data servers and Solid state Devices (SSDs) [47], and implementing incremental checkpointing [9]. Oliner, et al. present a strategy named coordinated checkpointing to skip unnecessary checkpoint/restart operations [46]. At runtime, the application requests a checkpoint and the system either allows or denies the checkpoint operation based on various system-wide heuristics, including disk or network usage and reliability information.

Liu, et al. present another reliability-aware optimal checkpoint/restart model for fault-tolerant applications in a MPP system environment [38]. In this model, the optimal time sequence of checkpoints is based on the theory of a stochastic renewal reward process. The authors state that, in general, this model can be used with any failure model. They tested their model with the Weibull distribution for time between failures based on their analysis of failure data associated with an HPC system at Lawrence Livermore National Lab (LLNL), for which about 80% of the data fit the Weibull distribution and 11% fit the exponential distribution.

Although there are various studies related to other effective fault tolerant techniques, still the most commonly used checkpoint/restart strategy is periodic checkpointing. And for periodic checkpoint/restart an optimal checkpoint interval, as presented by Daly [21], is based on the assumption of the exponential failure distribution. However, in a recent paper, Arunagiri, et al. present a model, based on Daly's model, that can be used to identify an optimal checkpoint interval with respect to number of I/O operations [12]. The authors observed that as the checkpoint interval increases, the trend related to the number of I/O operations follows that of execution time (ref: Figure 1.3). This model assumes that the failure distribution is exponential in nature and the results show that there is a region where by increasing the checkpoint interval to a value higher than the optimal interval identified by Daly's model, the number of I/O operations can be drastically reduced. Our research is based on this hypothesis.

2.2 Failure Distribution Pattern

Another important area related to this research is the distribution of failures in large-scale computing systems. The optimal checkpoint strategy proposed by Daly assumes that the failure distribution exhibits a Poisson single-component failure pattern [21]. Vaidya mentions that processor failures are assumed to have the Poisson failure distribution [62]. However, the community is divided in this regard. Various studies show that the Weibull

distribution is a better fit for failure distributions as compared to exponential or log-normal distributions [52], [32], [64], [50], [43]. In addition there is published literature stating that the gamma distribution is a good fit for failure models [34].

The main set-back in the study of failure patterns is the unavailability of failure data of MPP clusters. In this regard, Carnegie Melon University (CMU) took the initiative to create a repository of failure data known as the Computer Failure Data Repository (CFDR) [4]. CFDR has made available failure data from various clusters, the largest available failure data being from Los Alamos National Laboratory (LANL). LANL released failure data for 22 clusters collected over a period of nine years, from December 1996 to November 2005, which covers more than 23,000 outages. This data has been used in various statistical studies to identify failure patterns, identify the root cause of failures, and compute expected Mean Time Between Failures (MTBF) and repair [52], [34] or to help create failure prediction frameworks [25]. In [52], the authors perform a statistical analysis of the inter-arrival time of failures and found that both system-wide and node-wise, the Weibull and gamma distributions fit the failure data well. However, their study only provides a comparison of their log likelihood for exponential, Weibull, gamma and log-normal distributions, and concludes that the failure distribution is closer to the Weibull distribution. They do not perform any standardized goodness of fit test. Also, the authors only conducted the study of all failures occurring in the system. They have not classified the failures pertaining to any particular category. In our study we classified the data as hardware and software failures, and performed statistical curve-fitting analysis.

Kondo, et al. created a tool for comparative analysis of nine failure traces including LANL traces [34]. Their research shows that Weibull, log-normal and gamma distributions are better in mapping the availability and unavailability trends than the exponential distribution. However, there are three traces that fit exponential quite well.

Another widely analyzed source of failure data is the Reliability, Availability, and Serviceability (RAS) logs of the Blue Gene/P Intrepid system collected from January 2009 to August 2009. Taerat conducted a study of the RAS logs to compute an application's MTTI

based on the RAS messages [44]. RAS logs have various temporal and spatial redundant data. The authors consider only RAS logs that have severity levels labeled FAILURE or FATAL. Also they filter out and discard messages with temporal redundancy. The study attempts to strengthen the labeling of logs into FATAL and FAILURE logs by testing the messages and trying to identify only those that eventually lead to failure. The authors disregarded the messages that did not cause any failures. Their strategy included both best case dataset (only messages that will cause failures) and worst case dataset (both failure messages and undetermined messages included). As expected, the MTTI is greater for the best case and smaller for the worst case. As the time to repair (TTR) cannot be predetermined, the authors performed a sensitivity analysis of the MTTI by changing the TTR from five minutes to eight hours with a five-minute step size. The MTTI based on different TTRs was computed and presented graphically. Their results show that that the MTTI does not vary with a repair time higher than 20 minutes . In another study, Zheng performed a co-analysis of RAS logs along with Blue Gene/P job logs, to filter redundant data and then perform curve fitting on the filtered data to identify failure patterns [68]. Their results suggest that the Weibull distribution fits better than the exponential distribution for system-wide failure inter-arrival distributions based on Maximum Likelihood Estimation (MLE). They use a likelihood ratio test as the statistic to evaluate the effectiveness of these distributions. However, from the graphs presented, although we can see that the Weibull distribution fits better than the exponential distribution, visually the exponential distribution does not seem like a bad estimate. Alhough the Weibull distribution seems to provide a better estimate, the exponential distribution is a simpler distribution. If the error percentage of assuming an exponential over Weibull is small, for modeling reasons the use of a simple function is helpful and easy to work with.

Another dataset available in the CFDR is the log of hardware failures recorded on the MPP2 system (a 980-node HPC cluster) at Pacific Northwest National Laboratory (PNNL) [4]. Zarza used this data in work related to fault-tolerant routing for multiple permanent and non-permanent faults in HPC systems [66].

Yet another viewpoint exists with respect to the occurrence of failures. Some researchers believe that failures are not independent. In many failure conditions, like failures due to power outages, multiple nodes of the system may fail simultaneously. Also different applications running on the same node can fail due to a single failure condition. However, these failures, although separately logged, are dependent on one another. This is the main motivation behind the standpoint that failures are clustered. Hence, there are studies related to finding clusters in the failures. Hacker performed a study of clustered failures of data collected from two IBM Blue Gene systems at located at Rensselaer Polytechnic Institute (RPI) and at Ecole Polytechnique Federale de Lausanne (EPFL) in Lausanne, Switzerland [30]. He showed that it is possible to partition the system into sets of nodes ranked by reliability, which can be used to steer node allocation among scheduling queues and guide checkpointing strategies to reduce the probability and costs of failure. The authors initially identified dependencies, which appear as clusters or patterns in the data; then they replaced these clusters with a smaller number of events to act as an independent proxy for the dependent set of events; finally they measured the time between events in the resulting dataset to determine the statistical distribution of the elapsed time between independent, uncorrelated events and performed statistical fit. Their results show that the Weibull distribution fits better than the exponential distribution. Thanakornworakij, et al. performed a study on the effect of correlated failures on the reliability of systems [61]. They developed a model based on the Marshall-Olkin multivariate model with the Weibull distribution that derives the values for reliability, failure rate and MTTI. Their study shows that if components of the system have some dependency, the reliability decreases.

Most studies on failure distribution suggest that some dependence exists between failures. Thus, unlike the assumption made by Young [65] and Daly [21] that failure distribution is exponential in nature, exponential failure might not be a good estimate of failure distribution. In our study we examine the effectiveness of the analytical model based on the exponential distribution, Weibull distribution and the failure data available from various clusters.

Chapter 3

Methodology

To study the behavior of the number of checkpoint I/O operations and the execution time as a function of the checkpoint interval for a real HPC application, experiments were conducted with a popular HPC application on Ranger at TACC. The study was conducted in three parts.

- First, collect data related to checkpoint/restart operations such as the size of checkpoint data, checkpoint latency, execution time, and number of checkpoint I/O operations for an application running on Ranger.
- Second, analyze failure distributions from historic failure data to ascertain representative values of MTTI.
- Third, conduct a simulation study driven by data collected from real experimental runs and failure analysis.

Accordingly, this chapter is divided into three sections. In Section 3.1 we present the details of the experiments conducted on Ranger, while Sections 3.2 and 3.3 present the methodologies used to perform the analysis of failure distributions and conduct the simulation study.

3.1 HPC Experiments

The experiments conducted on Ranger used the RAxML and RAxML-Light applications, which are both described in Section 3.1.1 In Section 3.1.2, the details of the modifications

made to both of the applications are provided. Details of the experimental platform and test datasets have been presented in Section 3.1.3.

3.1.1 Application

The application selected for this study, RAxML, is an important application that is widely used and generally requires a long execution time. We used both RAxML (Randomized Axelerated Maximum Likelihood) and RAxML-Light. Both are community codes used for phylogenetic analysis. Phylogenetics is the study of evolutionary closeness among organisms. A phylogenetic tree is a mathematical structure representing the evolutionary history of a group of organisms or genes [49]. Given a set of aligned proteins of gene sequences, the aim of a phylogenetic study is to determine among all possible trees which one best describes the evolutionary relationships among the proteins. RAxML is used for this. It is a program for sequential and parallel Maximum Likelihood-based inference of large phylogenetic trees. It operates on both nucleotide and protein sequence alignments [53]. RAxML employs several heuristics to drastically reduce likelihood search times [49]. These heuristics include:

1. an initial starting tree under parsimony using random stepwise addition;
2. lazy subtree Rearrangements for branch swapping;
3. GTR + CAT (GTR with per site rate categories) model for evaluation of an inference tree instead of GTR + GAMMA; and
4. simulated annealing, which incorporates a cooling schedule and allows backward steps during the hill-climbing process.

The inputs to RAxML are [56]:

- -s option sequence file
- -t option starting tree (optional)

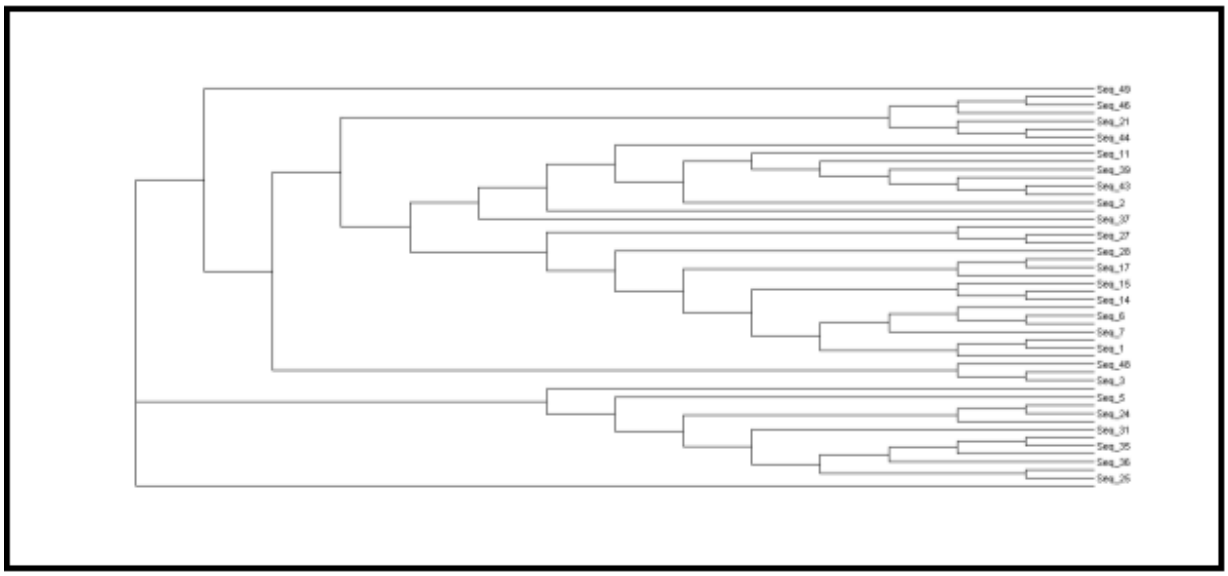


Figure 3.1: Output of Dendroscope

- -n option name of output file.
- -m option - substitution model (-m option)

RAxML begins the inference process by building a starting tree in PHYLIP format, which is done progressively, i.e., adding the sequences one by one in random order and inferring the best starting tree using the parsimony optimality criterion. Once the starting tree has been generated, the tree optimization process is carried out. RAxML performs standard sub-tree rearrangement (SPR) by subsequently removing all possible sub-trees from the current best tree and re-inserting them into neighboring branches up to a specified distance of nodes away. This process is repeated until there is no improvement in topology to be found. The best tree is written as a text file. The original algorithm is presented in a paper [53] and in a tutorial [49].

The output can be visualized using various visualization tools. However, the RAxML community advises the use of the Dendroscope tool [2]. A sample output of Dendroscope is shown above in Figure 3.1.

Checkpointing in RAxML is initiated and controlled by the external library DMTCP, which implements the coordinated checkpointing method by simultaneously suspended all worker threads or processes during checkpointing, i.e., while the master thread performs the checkpoint write [11]. Network data and kernel buffers are saved as a part of the checkpoint data along with the state of the application. At the startup of a new process, *dmtcp_checkpoint* injects *dmtcphijack.so* into the user's library and DMTCP adds a wrapper to some of the libc functions to make it aware of

- all forked child processes,
- all attempts to create remote processes, and
- all the parameters by which all sockets are created.

During a normal failure-free execution cycle, the following steps are performed:

- The *checkpoint manager thread* in each process waits for a new checkpoint request from the coordinator.
- Once, the checkpoint interval has elapsed, the coordinator sends a checkpoint request to all worker threads. The user threads suspend work and the socket buffers are drained and handshakes with peers are performed to discover the *globally unique ID* of the remote side of all sockets.
- The *connection information table* is written to disk. All user space memory and socket buffers are written to the checkpoint file along with a shell script, *dmtcp_restart_script.sh*, which contains all the commands needed to restart the program. The checkpoint file and restart script are written to persistent storage.
- Once the checkpoint file is written, the checkpoint write cycle is complete, the kernel buffers are refilled and user threads resume operation.

In the case of a failure and restart, a single DMTCP restart process is created for restoring each host. The sockets are recreated by referring to the *globally unique ID* in the

connection information table stored during checkpointing. Finally, the user processes are recreated on each host.

During our work with RAxML, we discovered that the PThreads version of RAxML 7.2.6 was the only version that had DMTCP checkpoint support and it could scale up to only one node (16 processors) [55]. Since we needed a checkpointing HPC application that could run on multiple nodes, we migrated to RAxML-Light 1.0.5, which has inherent support for checkpointing and can run on multiple nodes using MPI.

RAxML-Light is a reduced version of RAxML for inferring large trees under the CAT approximation and GAMMA model of rate heterogeneity [54]. It can infer trees using maximum likelihood given a starting tree from RAxML or the parsimonitor program or a checkpoint file. It cannot perform bootstraps, searches on multiple trees, computation of starting trees, which is done by RAxML. The tree inference procedure is the same as explained for RAxML. Once execution is complete it outputs the best tree, which can be visualized using the Dendroscope program

RAxML-Light is optimized for running on clusters and implements non-periodic checkpoint/restart. A checkpoint operation is performed after each Subtree Pruning and Regrafting (SPR) tree-optimization cycle. The inputs to RAxML-Light are:

- -s option sequence file
- -t option starting tree
- -R option checkpoint file
- -n option name of output file
- -m option substitution model

3.1.2 Modifications to RAxML and RAxML-Light

To collect checkpoint/restart data and to understand the checkpoint/restart behavior of the application, it was required to execute the program in the presence of failures. This

translates to unpredictable and extremely long executions of the application, which was not feasible. To address this issue we extended RAxML and RAxML Light in the following ways to induce simulated failures:

1. input file with failure times (in seconds),
2. simulation of a failure (checkpoint file write) at the times specified by the new input file, and
3. simulation of a restart (checkpoint file read) after each simulated failure.

Another important modification caused checkpoints to be periodic, i.e., instead of being based on completion of an SPR cycle, they were based on elapsed execution time. This was necessary as most checkpoint strategies are based on performing checkpoint at a particular time.

In addition, during simulations we needed to collect the following checkpoint/restart data, which are required to study the performance of the two checkpoint models on real-life application.

1. checkpoint latency for each checkpoint operation,
2. amount of data transferred during each checkpoint read or write operation,
3. total execution time required with and without restart,
4. total rework time at each restart,
5. total number of checkpoint writes,
6. number of restarts, which is the same as the number of failures, and
7. restart time or the time taken for the application to restore the state while restarting from each failure.

In addition, data not directly related to checkpoint/restart, such as the start and the end time of each productive I/O operation and the count of productive I/O operations was required to gain a better understanding of, and thus characterize, the RAxML application.

We considered the following two tools, both transparent to the user, to determine the I/O characteristics of applications executed on Ranger. As described below, they were insufficient for collecting the information required for our research (e.g., checkpoint latency) and, thus, we had to add logging functions to the codes.

- TACC_stats provides various data regarding a job's execution, e.g., queue wait time, run time, system idle time, I/O wait time, memory usage, amount of data read in bytes, and amount of data written in bytes. It does not provide start and end times of checkpoint read and write operations and the amount of data transferred during each I/O operation, which is needed to compute checkpoint latency.

- Darshan, an I/O characterization tool from Argonne National Lab [16], can be used to determine application I/O behavior. Darshan is implemented as a set of libraries that are linked dynamically. It does not capture a complete I/O trace; rather, it captures a trace of MPI-IO routine calls. This functionality is realized by inserting wrapper functions in the profiling interface to MPI (PMPI) and POSIX routines, which maintain a record for each file opened (when first opened, when finally closed, and the total amount of data transferred). Darshan does not provide the information we require regarding each and every checkpoint file read and write.

To add logging and failure-inducing functions to the code, we modified the external library, DMTCP, which performs RAxML's checkpointing. The logging functions log the wall-clock start time (in milliseconds), wall-clock end time (in milliseconds), amount of data transferred (in bytes), and type of operation performed, e.g., Checkpoint Write (CW), Restart Read (RR), Productive Write (PW) or Productive Read (PR). They perform two basic tasks: (i) whenever an event from the specified set of trigger events occurs, the wall-clock time is stored into a memory buffer and (ii) the buffer is written to a file when one or

more of the following conditions occur: the buffer becomes full, a certain pre-specified time interval has passed, or the program completes its calculations. These functions were added as wrappers to the original read write operations of DMTCP so that whenever a read or write operation occurs, the data is collected.

The failure-inducing functions were added to the DMTCP library, which generates a restart script during each checkpoint write operation. This script contains information to restart the application using the latest checkpoint file. When a failure occurs, the associated processes terminate. However, a process can be restarted by running its restart script (*dmtcp_restart_script.sh*) using the latest checkpoint file. Our research required the capability of injecting failures events and, therefore, we simulated failure events within DMTCP. This was done by interrupting the application logically and not terminating it as in real failures. When interrupted a process reads the checkpoint file and resumes operation. The start time of each injected failure was read from in an external text document.

Since RAxML-Light is a reduced version of the original RAxML, the changes made to DMTCP for RAxML were directly migrated to RAxML-Light with minor syntactic changes. The overall semantic behavior was preserved. However, since the analytical models we are using are based on time-periodic checkpointing, the RAxML-Light code was modified to simulate time-periodic checkpoint writes. Instead of RAxML-Light writing a checkpoint file to persistent storage after each SPR cycle, it was changed to store that checkpoint state in memory and write that file to disk only after a specified checkpoint interval had elapsed. To do this, after each SPR cycle, the checkpoint state was saved in memory but the actual write to disk was performed later. The time period between checkpoints, i.e., the checkpoint interval was an input parameter.

3.1.3 Experimental Platform

The modified codes were tested at The University of Texas at El Paso (UTEP) before being used for experiments at TACC. The systems used at UTEP and TACC are described below.

Figure 3.2: Ranger at Texas Advanced Computing Center

1. **UTEP Test Bed:**

   - Dell PowerEdge SC 1420 computer with two Intel Xeon RK80546KG0721M processors capable of Hyper-threading running CentOS 5.3
   - 30 GB disk storage

2. **TACC MPP System, Ranger:**

   - 3,936 16-way SMP compute nodes (15,744 four-core AMD Opteron processors for a total of 62,976 compute cores)
   - 123 TB RAM
   - 1.7 PB disk storage

- 579 TFLOPS peak performance

Test datasets were downloaded from the RAxML website used for the 2007 Supercomputing paper on parallelizing RAxML on the IBM BlueGene/L [1]. Also a large dataset was downloaded from the iPTOL project website [5] for scalability testing; this dataset contained 34,584 taxa with 1,303 columns. The results of these experiments are reported in Section 4.

3.2 Failure Data Analysis

In an effort to define a realistic value for the node MTTI parameter of the models worked with in this research (and the simulations that we conducted, which are described in Section 4) and to verify whether the failure distribution is indeed exponential, as is assumed by the models, or Weibull, suggested by different literatures, we analyzed the historic failure data made available by Los Alamos National Laboratory (LANL) in 2006. The LANL data consists of records of cluster node outages, workload logs and error logs of 22 clusters collected from December 1996 through November 2005. The data was collected as a remedy database, where the system administrators of the 22 clusters updated details of the node failures that required system administrative intervention for resolution. The data has been classified into five large categories: human error (149 errors), environment (357 errors), network failure (420 failures), software failure (5,302 errors), and hardware failure (14,291 failures). There were some failures that could not be classified and were recorded under the unresolved category (3,094 failures). As shown, the two major contributors of failures were hardware and software failures. Accordingly, we conducted a study of the hardware and software failure inter-arrival time to find which statistical distribution best describes the failure pattern of each system, and to define a realistic value for node MTTI.

3.2.1 Preprocessing of the Failure Logs

In order to study the pattern of inter-arrival time of failures, we had to pre-process the logs to infer inter-arrival times. Since the log files contain the time of every failure for each node of a system, we were able to compute the time since the last failure, which is derived data, for a variety of failures. The program for performing this computation is presented in the appendix of [31]. The log was pre-processed to compute the following derived data:

1. Time since last node repair,
2. Time since last node upgrade,
3. Time since last node software failure.
4. Time since last node maintenance,
5. Time since last system repair,
6. Time since last system upgrade,
7. Time since last system software failure, and
8. Time since last system maintenance.

Since, the hardware and software failures were the largest contributors to failures in the clusters, this data was sub-divided into hardware and software failures. Errors like human errors and facilities errors cannot be controlled and are not predictable based on environmental factors, thus, associated data was not considered. Once the data preprocessing was completed, statistical curve fitting tests were carried out on the available data.

3.2.2 Statistical Distribution Fitting

As discussed earlier, analysis of the failure distribution could potentially be helpful for estimating the checkpoint interval and reducing the number of checkpoint I/O operations

of an HPC application. For time-critical applications, the optimal checkpoint interval that provides minimum execution time is given by Daly's model [21] and several others [37] [46] [38] for systems with failure distributions that fit exponential and Weibull distributions, respectively. Analysis of historic failure data helps ascertain the failure patterns historically exhibited in existing or retired systems and provides a good starting point for investigating failure patterns in current systems.

To perform the analysis, the following five tasks were performed.

1. Generate three histograms for a system: one of all hardware and software failures, one of only hardware failures and one of only software failures.

2. Examine how well these failures fit well-known statistical distributions referred to in reliability studies, namely exponential and Weibull.

3. Perform curve fitting using the fitdistrplus package of R; the Maximum-Likelihood Estimation was used as the method to fit distributions.

4. For each distribution, validate the analysis using the three commonly used goodness-of-fit tests, i.e., Kolmogorov-Smirnov(K-S) [41], Anderson-Darling(A-D) [10] and Cramer-von-Mises(CvM) [22].

Using the LANL logs, we also analyzed the data for mean-time-to-interrupt (MTTI); this data was for use in our simulation studies. We only considered the hardware errors for this estimation; the software errors are mainly related to either disk or I/O drivers, software upgrades or other unknown software failures, and, generally, it is not possible to predict a software failure. During the lifetime of a system, the software may be upgraded or replaced if required, hence, the trend and type of software failures can change during the lifetime of a system. However, the hardware of the system generally remains unchanged during the system lifetime and some hardware errors can be predicted or speculated based on the environmental conditions. The analysis of the MTTI was carried out for two LANL systems (systems 18 and 19). These systems were selected based on two factors:

1. They had the largest number of nodes, i.e., 1,024 nodes, each with four processors.

2. They had large failure counts. System 18 had 2,967 hardware failures and system 19 has 2,549 hardware failures reported (excluding maintenance activities).

We used two methods for determining the MTTI for these two systems: MTTI based on all the failure data and MTTI based on a subset, i.e., n years, of data. The first method employed the following process:

1. For each node, all times to interrupt (TTIs) were computed and the average of these values was taken to be the MTTI of that particular node.

2. For each node with no failures recorded, the largest reported TTI for all other nodes was considered to be its MTTI.

3. Once the MTTI for each node was computed, the MTTI of any node of the system was computed as the average of the computed node MTTIs. This was possible because all the nodes had the same number of processors, i.e., four, and similar memory and CPU configurations.

4. Once the average node MTTI was computed, the system MTTI was computed as $MTTI_{system} = \frac{MTTI_{node}}{Number of Nodes}$.

The second method, MTTI based on n years of failure data, only considered those failures that occurred within n years of installing the system. It employed the same process described above. This analysis was done to see how a system's MTTI changes over time. This also helps us understand how well we can estimate the MTTI of a system during its initial deployment period and after this period. For the analysis of failure data during the initial period of deployment, we used n = 1, 2, and 3. The results of our analyses are presented in Section 4.3.

3.3 Simulation Studies

The data collected from the experiments on Ranger and the failure analysis were used in simulation studies performed to analyze the impact of the checkpoint interval on the number of checkpoint I/O operations and execution time. The simulator, which was coded in Java and is presented in [31], is designed as follows. The input for the simulations discussed in this thesis are noted in parantheses.

1. Input:

   (a) Job list (which can contain multiple jobs), i.e., an input file where each job entry specifies: the job arrival time, solution time (in minutes), checkpoint interval (in minutes), restart overhead (in minutes), checkpoint latency (in minutes) and number of nodes to run required to run the application (the characteristics of RAxML-Light were used to define these parameters and the checkpoint latency was based on our HPC experiments run on 100 nodes of Ranger);

   (b) Arrival times of the jobs in the job list multiple jobs can be concurrently executed;

      i. A job's arrival time equates to its simulation start time it is the time when the job is expected to start running, expressed as the time elapsed since the allocated nodes were installed (in minutes), e.g., if a job is supposed to run on x nodes of system Y, which were installed at time T_s, with a start time of t, then this implies the job will run from time $T_s + t$ and is expected to face failures that occur on those x nodes of system Y that occur after that time;

   (c) Number of nodes in the simulated system (1,024 as per LANL systems 18 and 19); and

   (d) External file that provides system failure information (failure files for LANL systems 18 and 19) this includes times at which failures occur and repair times.

2. Process: During a simulation,

   (a) The above-mentioned inputs are read;

   (b) Based on the inputs, the scheduler is initialized to the earliest start time of the jobs waiting in the job queue. Initially all nodes are expected to be available. The list of available nodes of the system is maintained for scheduling purposes; as nodes are allocated for each job, this list is modified accordingly.

   (c) All jobs are maintained in a job queue. The scheduler reads the jobs from the job queue in the order of arrival time from the job queue. If the number of nodes requested by a job is not available (i.e., the nodes are busy executing some other jobs), the job is kept in the wait priority queue, the priority being based on job arrival time. Thus, a job with earliest arrival time is expected to have maximum priority and run immediately when nodes are available. Once nodes are available, the top priority job in the wait queue that requires the available number of nodes or less starts (or restarts) operation.

   (d) The scheduler maintains a heap of events and performs operations based on the event type. The event heap is a min heap. Thus the operation that will occur earliest will always be polled by the scheduler.

   (e) Initially job each job a new event in created with time equal to the job arrival time and type equal to job scheduled. The failures start time associated with each node failure is added as a new event with failure start type and time equal to the failure start time (read from inputs).

   (f) The scheduler polls the event queue till there are no more events or all jobs have finished. The events that are encountered during the lifetime of the execution are as follows:

      i. Event job scheduled: If there are available resources for the job to run, then the job is scheduled to run. Otherwise the job is added to the wait priority

queue. For jobs that can run immediately, the $time_{now}$ is updated with time to perform restart. A new event with type as checkpoint start and time equal to $time_{now} + checkpointinterval$ for that job is created. This event is added to the event heap.

ii. Event checkpoint start: The execution time is updated by adding the checkpoint interval. The latency is added to present time to compute the checkpoint end time. A new event is created (Checkpoint End) with the time equal to the checkpoint end time and is included in the event queue.

iii. Event checkpoint finish: The latency is added to the execution time of the job. The last checkpoint time is updated to the checkpoint start perform time. The next time to perform checkpoint is computed as the next checkpoint start time. If the computation to be done is less than the checkpoint interval, i.e., the job will finish before next checkpoint write operation, a new event is created as Job finished with time equal to the time to finish the job. Otherwise a new Checkpoint Start event with the time equal to the next checkpoint start time is created. In both the cases, the created event is added to the event heap.

iv. Event failure start: This implies a node that is running has failed. Thus, the corresponding job will fail. All other nodes are freed and put in the list of available nodes. The job is put into the wait priority queue. The scheduler is invoked to see if there is any job that can start execution. The repair time for this failure is added to the failure start time to compute the failure end time. A new event is created (Failure End) with the time equal to the failure end time and is put into the event heap.

v. Event failure end: This event means that a node has been repaired. This event is put into the event heap. The repaired node is added to the list of available nodes. The scheduler is invoked to see whether it can schedule any new jobs.

vi. Event job finished: The duration elapsed between the time now and the last job start time is added to the execution time. The job is removed from the job queue. The nodes are freed and added to list of available resources. The count for finished jobs is increased. The scheduler is invoked to see whether any job can be rescheduled with the newly available resources.

(g) For each job, the checkpoint interval, along with the expected latency, is specified in its entry in the job list. When the checkpoint interval has elapsed, the job writes a single checkpoint file to disk and its execution time is incremented by the expected latency to write the file.

3. Output: Once all jobs are completed, the simulation is complete. The simulator then outputs:

   - execution time of each job,
   - number of checkpoint write operations, and
   - number of failures that occurred.

Chapter 4

Results

As stated earlier, the main objectives of our research are to:

- Study the variation of the execution time and the number of I/O operations as a function of the checkpoint interval.
- Investigate patterns in historic failure data to provide information for realistic values of expected node and system MTTI and for scheduling of checkpoints.
- Using historic failure data, conduct a simulation-based study of a selected HPC application to determine the percentage error of the estimates of execution time and number of I/O operations made by the two analytical models.

A secondary objective of this research is to predict the MTTI of the system based on n years of historic failure data. The objective is to determine the percentage error of the estimates of execution time and number of I/O operations made by the two analytical models using this predicted MTTI in their computation.

As described previously, both a study of HPC execution of RAxML and RAxML-Light, and a simulation-based study that uses historical failure data were employed to determine the percentage error of the execution time and number of checkpoint I/O operations estimated by the two analytical models under study. The methodology used to conduct the experiments that achieve these objectives was described in the last chapter. This chapter presents the results of these experiments. The results of our analysis of failure data is presented in Section 4.1, which is followed by the study the variation of the execution time and number of checkpoint I/O operations as a function checkpoint interval for RAxML and

RAxML-Light executed on Ranger, which is presented in Section 4.2. In this study simulated failures with exponential distribution were injected into application runs. To study the effects of running RAxML-Light for longer durations on a larger number of nodes, with failure events occurring as recorded in the LANL failure remedy database, studies were performed on a Java simulator using the checkpoint/restart parameters collected from running RAxML-Light on Ranger. The results of this simulation study are presented in Section 4.3.

4.1 Analysis of Failure Data

As described in Section 3.2, the hardware and software failure data from LANL, taken together and individually, were analyzed using R to find statistical distributions that best fit the data; *fitdistrplus* was employed for curve fitting. The Weibull and exponential distributions were considered since they are the two most popular distributions associated in terms of trends of failure inter-arrival times. The process used to analyze the selected hardware and software failure data follows:

1. Compute the following inter-arrival times:

   (a) Inter-arrival time between any failures, i.e., time since last failure,

   (b) Inter-arrival time between hardware failures, i.e., time since last repair; and

   (c) Inter-arrival time between software failures, i.e., time since last software failure.

2. Perform curve-fitting tests, using Maximum Likelihood test, for hardware failures, software failure and hardware-software failures taken together for each of the 22 systems.

3. Conduct the goodness-of-fit tests using K-S, A-D and C-v-M tests.

4. Repeat the same curve-fitting tests for each node of any system that had more than 30 failures. The system considered was system 2, in which 25 out of 49 nodes had more

than 30 failures. For the other 21 systems there were very few nodes that qualified for this test and, hence, they were not considered.

1. **Hardware-Software System-wide Failure Analysis:**

   For each cluster or system, the results of performing statistical distribution fit for all hardware and software failures taken together have been provided. The shape and scale factor for the best fit Weibull distribution and the rate for the exponential distribution have been presented. Finally, the measure of the goodness-of-fit has been provided. The 3 goodness-of-fit tests (mentioned in section 3.2.2) are run and if the result fails the test it is mentioned as R (Rejected). Otherwise if the fit is not rejected, it has been shown with NR. For system 17, these tests were not performed as failure record contained null values.

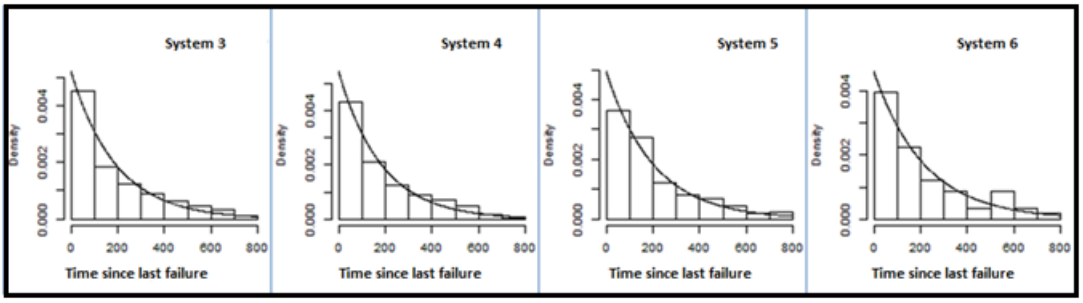
   The results are provided in Table 4.1.

   As shown by the data in Table 4.1, 9 out of 22 systems conform to the Weibull distribution whereas 4 out of 22 systems conform to exponential distribution. For systems 3, 4, 5, and 6, which conform to both Weibull distribution and exponential distribution, the shape factor of the Weibull distribution is close to 1. This is understandable since it is known that a Weibull distribution with a shape factor = 1 is an exponential distribution.

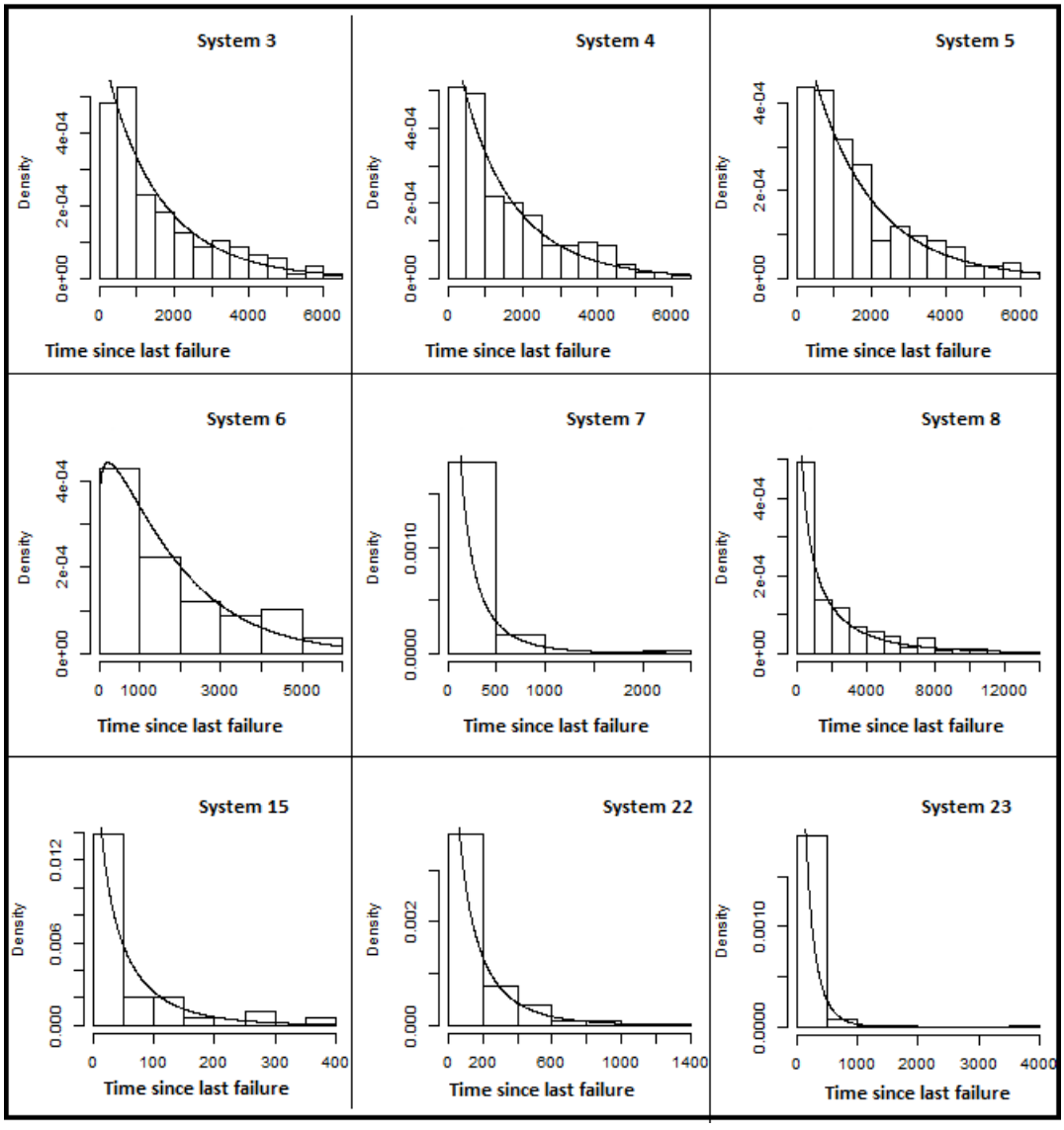
   Thus, the study conformed to the popular belief that a Weibull distribution better estimates the failure distribution than does an exponential distribution. Note, however, that an exponential distribution also matched in 45% of the cases; hence, the hypothesis that a failure distribution can be exponential cannot be disregarded completely. Figure 4.1a above present the curve fitting graphs of those elements for which any of the goodness-of-fit test did not reject exponential distribution. Figure 4.1b below, show the plots of graphs for which Weibull distribution is not rejected. The plots show that the inter-arrival time conforms closely to statistical distributions like

Table 4.1: Hardware-Software System-wide Failure Analysis

Cluster #	Goodness-of-fit for exponential distribution			Goodness-of-fit for Weibull distribution			Shape	Scale	Mean
	K-S	A-D	C-v-M	K-S	A-D	C-v-M			
2	R	R	R	R	R	R	0.65	133.78	5.10E-03
3	NR	NR	NR	NR	R	R	0.98	1,529.97	6.50E-04
4	NR	R	NR	NR	R	R	0.96	1,447.75	6.80E-04
5	NR	R	NR	NR	R	R	0.99	1,619.20	6.20E-04
6	NR	NR	NR	NR	NR	R	1.11	1,805.67	5.80E-04
7	R	R	R	NR	R	R	0.65	135.30	5.20E-03
8	R	R	R	NR	R	R	0.68	1,646.70	4.80E-04
9	R	R	R	R	R	R	0.79	2,400.88	3.80E-04
10	R	R	R	R	R	R	1.01	2,863.33	3.50E-04
11	R	R	R	R	R	R	0.72	2,105.15	4.10E-04
12	R	R	R	R	R	R	0.76	2,222.76	4.00E-04
13	R	R	R	R	R	R	1.02	3,008.12	3.30E-04
14	R	R	R	R	R	R	0.68	1,468.95	5.70E-04
15	R	R	R	NR	R	R	0.79	50.36	1.70E-02
16	R	R	R	R	R	R	0.64	102.97	6.20E-03
18	R	R	R	R	R	R	0.80	1809.91	5.00E-04
19	R	R	R	R	R	R	0.83	1,783.54	5.20E-04
20	R	R	R	R	R	R	0.57	688.09	9.60E-04
21	R	R	R	R	R	R	0.61	330.98	2.30E-03
22	R	R	R	NR	R	R	0.77	131.85	6.40E-03
23	NR	R	R	NR	R	R	0.81	137.73	6.40E-03



(a) Exponential Distribution was "Not Rejected"



(b) Weibull Distribution was"Not Rejected"

Figure 4.1: System Hardware-Software Failure Inter-arrival Time Distributions

exponential and Weibull. The probability of occurrence of failures is high after repair from a failure.

2. **Hardware Only System-wide Failure Analysis:**

   As described in Section 4.1, the hardware failures for all 22 LANL systems were taken collectively and curve fitting was carried out for each system. As system 7 had less than 30 hardware failures and, thus, is not included in this analysis. Also system 17 was not considered, because the LANL log entries for it was incomplete. In table 4.2 the result of curve fitting for hardware failures have been presented. The table contains same information as in the Table 4.1 for Hardware-Software system-wide failure analysis.

   Figures 4.2a and 4.2b, present the plots for those systems for which the curve fitting results were favourable.

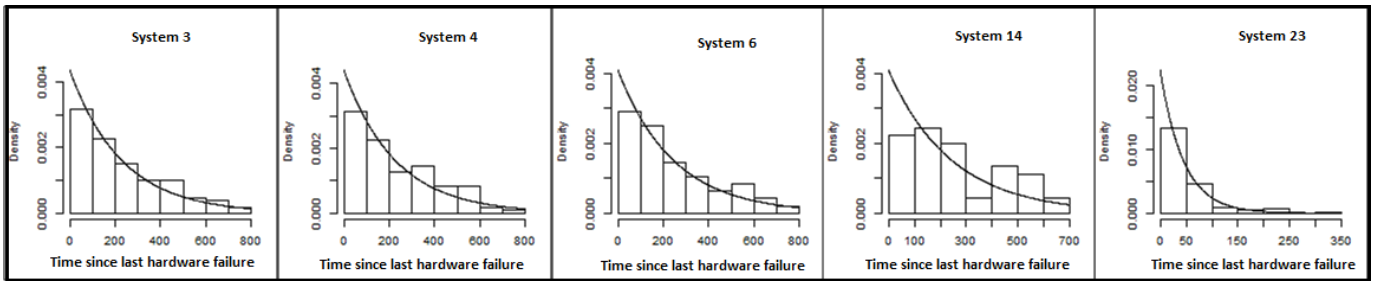
   In analyzing only the hardware failure data, we found that the failures of ten systems conformed to the Weibull distribution, i.e., they were not rejected by the curve fitting, and five conformed to the exponential distribution. Overall, as compared to the combined hardware and software failures, the hardware failures of these 22 systems more closely follow the Weibull distribution. Accordingly, a better prediction of hardware errors is possible. The rate of failure varies from 0.000034 to 0.0017. The shape factor of the reference Weibull distribution varies from 0.53 to 1.27. Figures 4.2a and 4.2b show the plots for those systems, which were not rejected by any of the goodness of fit tests, for exponential distribution and Weibull distribution respectively. For 3 out of 20 systems, higher order AD test was successful and for 1 out of 20 cases the CvM test gave positive results. Also the CvM test was successful only for exponential distribution and not for Weibull. From the figures (4.2a and 4.2b) of system 6, we can visually conclude that exponential distribution is a better fit than Weibull distribution.

Table 4.2: Hardware System-wide Failure Analysis

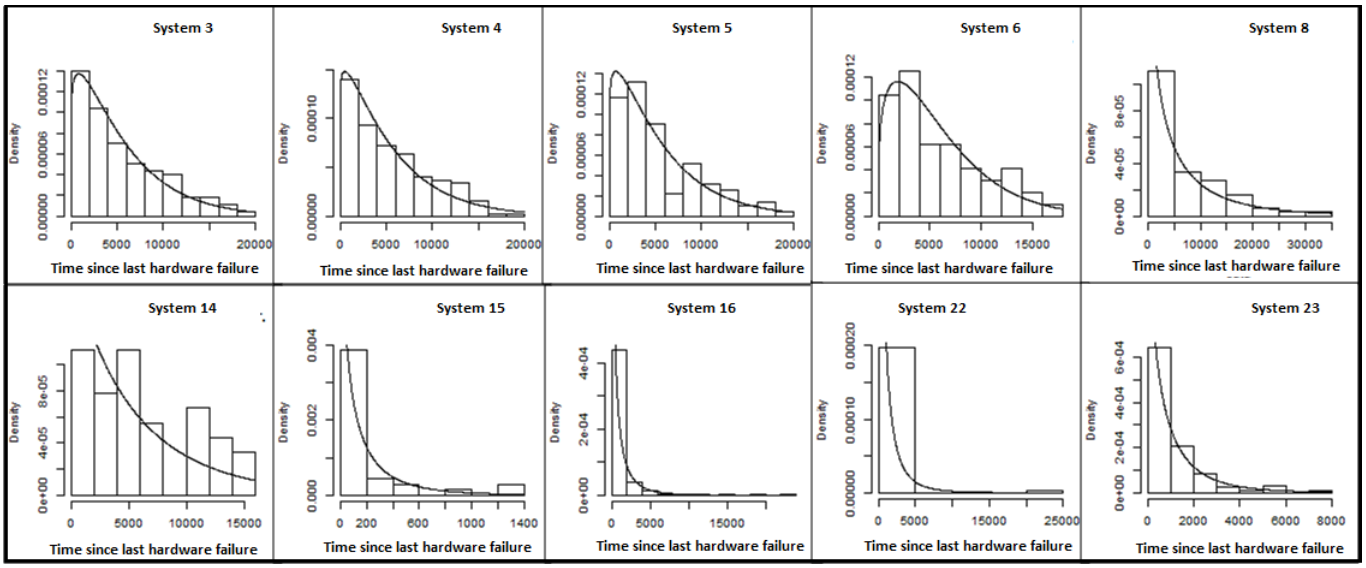
Cluster #	Goodness-of-fit for exponential distribution			Goodness-of-fit for Weibull distribution			Shape	Scale	Mean
	K-S	A-D	C-v-M	K-S	A-D	C-v-M			
2	R	R	R	R	R	R	0.69	679.74	3.70E-04
3	NR	R	R	NR	NR	R	1.13	5,738.37	6.00E-05
4	NR	R	R	NR	R	R	1.08	5,591.28	6.00E-05
5	R	R	R	NR	R	R	1.11	5,611.77	6.00E-05
6	NR	NR	NR	NR	NR	R	1.27	6,344.28	5.70E-05
8	R	R	R	NR	R	R	0.72	5,842.76	4.70E-05
9	R	R	R	R	R	R	0.80	7,393.54	4.00E-05
10	R	R	R	R	R	R	0.96	8,319.34	4.00E-05
11	R	R	R	R	R	R	0.76	6,833.16	4.30E-05
12	R	R	R	R	R	R	0.72	6,279.03	4.70E-05
13	R	R	R	R	R	R	0.98	8,842.43	3.70E-05
14	NR	R	NR	NR	R	R	0.96	5,797.75	5.70E-05
15	R	R	R	NR	R	R	0.71	150.95	1.70E-03
16	R	R	R	NR	R	R	0.60	699.58	3.00E-04
18	R	R	R	R	R	R	0.82	6,441.64	4.70E-05
19	R	R	R	R	R	R	0.93	6,540.88	5.00E-05
20	R	R	R	R	R	R	0.53	2,353.69	8.70E-05
21	R	R	R	R	R	R	0.54	921.71	2.40E-04
22	R	R	R	NR	NR	R	0.61	692.39	3.00E-04
23	NR	R	R	NR	NR	R	0.78	927.82	3.10E-04

3. **Software Only System-wide Failure Analysis:**

   Similarly for analysing software failures of all 22 LANL systems taken collectively, curve fitting was carried out for each system. Since software failures are much lesser in number than hardware failures as described in Section 3.2 6 out of 22 systems of LANL(6,12,13,14,15,21) had less than 30 hardware failures and, thus, were not included in this analysis. Also system 17 was not considered, for similar reasons state above. In table 4.3 the result of curve fitting for software failures have been presented.



(a) Exponential Distribution was "Not Rejected"



(b) Weibull Distribution was"Not Rejected"

Figure 4.2: System Hardware Failure Inter-arrival Time Distributions

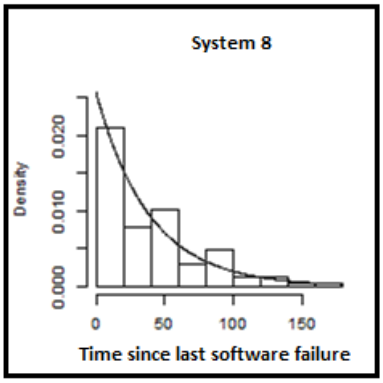
The table contains the results of the goodness of fit test for KS, AD and CvM tests, described in section 3.2.2. Also the shape and scale factor of the best fitted Weibull distribution and the rate parameter of the best fit exponential distribution has also been presented. Figures 4.3a and 4.3b, present the plots for those systems for which the curve fitting results were "Not Rejected" by any of the three goodness of fit tests.

Curve fitting could only be carried out on fifteen of twenty-two systems, seven systems did not have failures greater than 30. Analysis of the software failure data showed

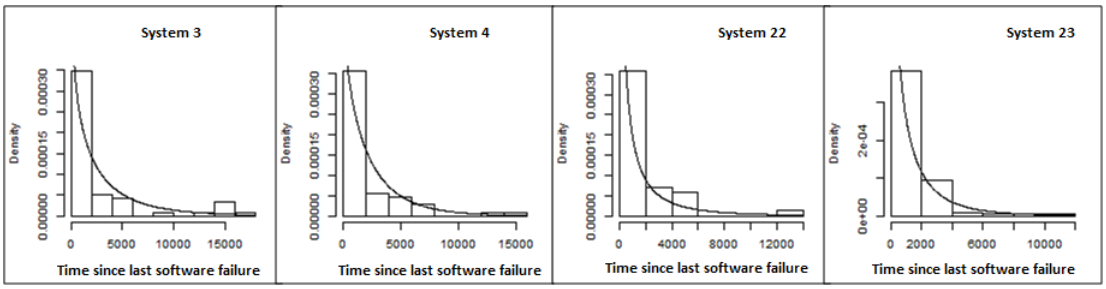
Table 4.3: Software System-wide Failure Analysis

Cluster #	Goodness-of-fit for exponential distribution			Goodness-of-fit for Weibull distribution			Shape	Scale	Mean
	K-S	A-D	C-v-M	K-S	A-D	C-v-M			
2	R	R	R	R	R	R	0.51	644.92	2.20E-04
3	R	R	R	R	R	R	0.80	2,839.01	1.00E-04
4	R	R	R	NR	R	R	0.90	2,358.38	1.30E-04
5	R	R	R	NR	R	R	0.58	3,576.46	6.30E-05
7	R	R	R	R	R	R	0.51	321.79	4.00E-04
8	NR	R	R	R	R	R	0.82	8,551.78	3.70E-05
9	R	R	R	R	R	R	0.94	11,230.22	2.90E-05
10	R	R	R	R	R	R	2.95	15,569.62	2.30E-05
11	R	R	R	R	R	R	1.55	13,002.56	2.70E-05
16	R	R	R	R	R	R	0.59	634.4863	3.10E-04
18	R	R	R	R	R	R	0.81	7,250.36	4.00E-05
19	R	R	R	R	R	R	0.68	6,065.5	4.30E-05
20	R	R	R	R	R	R	0.65	4,672.14	5.30E-05
22	R	R	R	NR	R	R	0.55	909.01	2.10E-04
23	R	R	R	NR	NR	R	0.68	1,097.17	2.40E-04

that four out of fifteen systems were not rejected for the Weibull distribution and one was not rejected for the exponential distribution; and notice that none of the systems passed the higher order tests for goodness of fit. Unlike hardware failures, less software failures conformed to the well-known failure distributions. The data for system 3, which fits the Weibull distribution when hardware and software failures were considered together, did not fit this distribution when software failures were considered alone. From the plots we can see that even for those systems for which the not rejected by either Weibull or exponential distribution, does not visually comparable to the hardware data curves. The rate estimated from exponential distribution varies widely from 0.000023 to 0.0004. The shape factor for Weibull distribution is less than 1 for thirteen out of fifteen cases.



(a) Exponential Distribution was "Not Rejected"



(b) Weibull Distribution was"Not Rejected"

Figure 4.3: System Software Failure Inter-arrival Time Distributions

4. **Estimation of MTTI**

To estimate the MTTI of a system, the average MTTI of the nodes was computed. Since the date of installation of every node on the studied LANL systems was provided along with the timestamp of the start of every failure, the computation of the MTBF of each node was easily completed. If there were no failure records for a node, that node was assumed to be alive for the entire time duration. Once the node MTTI of each node of a system was computed, an average node MTTI was computed. Finally, the system MTTI was computed as $SMTTI = \frac{NodeMTTI}{Numberof Nodes}$.

The systems with same number of nodes and similar configurations, i.e., systems 3, 4 and 5; systems 9, 10 and 11; and systems 18 and 19, have similar MTTIs. Systems 2 and 15 have a largest number of processors per node, i.e., 128 and 256, respectively.

Table 4.4: MTTI computed from LANL data

System Number	Number of Nodes	Number of Processors per Node	Average Node MTTI (days)	System MTTI (days)
2	49	80 and 128	44	0.89
3	128	4	358	2.79
4	128	4	380	2.97
5	128	4	369	2.88
6	32	4	374	11.68
7	1	8	182	182.00
8	164	2	943	5.75
9	256	2	644	2.52
10	256	2	623	2.43
11	256	2	624	2.44
12	512	2	694	1.36
13	256	2	654	2.56
14	128	2	527	4.12
15	1	256	8	8.22
16	16	128	55	3.44
18	1024	4	473	0.46
19	1024	4	472	0.46
20	512	4	528	1.03
21	128	4	134	1.04
22	1	4	47	46.92
23	5	32 and128	76	15.2

Accordingly, systems with a smaller number of nodes t, with the exception of system 2, have a greater MTTI. Confirming an observation in [52], the MTTIs of systems 2, 15, 16 and 23, which are NUMA-based systems, are quite different from that of the other systems, which are SMP-based clusters. So we can see that memory access patterns have an impact of the MTTI of the systems.

4.2 Experiments on Ranger

As explained in Section 3.1, RAxML and RAxMLLight were executed on Ranger at TACC. Since RAxML (with DMTCP checkpointing) could not run on more than one node, we only ran it to collect information that would allow us to profile the I/O of RAxML. In contrast, RAxML-Light was used to study the variation of execution time and the number of I/O operations as a function of the checkpoint interval.

4.2.1 RAxML Experiments

We executed four runs of RAxML on one node of Ranger to determine RAxML's I/O characteristics. We ran with two different checkpoint interval (16.67 minutes and 30 minutes) respectively to study the I/O statistics. The results have been presented in Table 4.5. Also the plots for I/O data is shown in Figure 4.4.

Table 4.5: Parameters for RAxML Experiment

Checkpoint Interval (in minutes)	Number of checkpoint I/O Operations	Number of productive I/O operations
16.67	7	16
30.00	5	16

As shown in Figure 4.4, RAxML's productive I/O (red) is periodic and bursty, while its defensive (checkpoint) I/O (green) is periodic. For the problem size of 500 gene sequences (or taxa), each of length 5000, the volume of defensive I/O was about 22MB, whereas the volume of the productive I/O was 4KB. Thus, in the case of RAxML, the defensive I/O is the main user of I/O utilization.

4.2.2 RAxML-Light Experiments

Since RAxML 7.2.6 cannot run on more than one node, RAxML-Light 1.0.6 was used to experiment with larger node counts. In all the test runs i.e. 3 run for each checkpoint

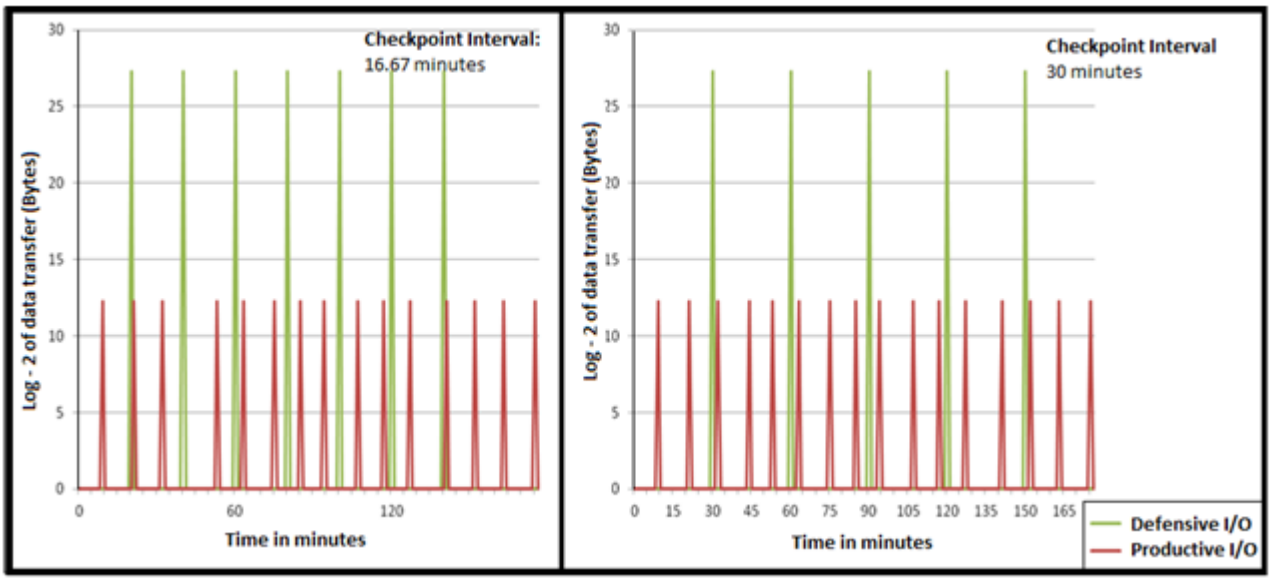


Figure 4.4: RAxML on Ranger: I/O Characteristics of RAxML 7.2.6

interval we employed four nodes, i.e., 16 processors; however, we ran a couple of short duration experiments (less than 5 hours) on 100 nodes to measure the checkpoint latency of RAxML-Light on larger number of nodes. This was conducted to compute the average latency in case we run an application on 100 nodes which would indeed help in our simulation studies for longer durations (presented in section 4.3).

This experiment was conducted to observe the variation of the number of I/O operations and the execution time as a function of the checkpoint interval for RAxML-Light executing on Ranger.

Experiments in this set were conducted by simulating failure and restart during the execution of RAxML-Light on Ranger. The simulated failures assume an exponential failure distribution with parameter values of node MTTI equal to 24 hours and 32 hours. This assumption was made to ensure that we have a reasonable number of failures during the execution of the application with solution time of 20 hours. The number of nodes in this experiment was 4 and therefore the system MTTI corresponding to node MTTI of 24 hours and 32 hours was 6 hours and 8 hours, respectively. Although this seems unrealistic

at the outset, it seems reasonable considering the fact that the next generation Exascale systems are predicted to have system MTTI about 1 hour. This is due to the fact that millions of components work together in such large scale systems. Since the goal of this study is to investigate the variation of the number of I/O operations and the execution time as a function of the checkpoint interval, the choice of and the range of values of the independent variable, the checkpoint interval is important. For these experiments we chose to use $0.5\tau_{opt}, \tau_{opt}, 2\tau_{opt}, and 4\tau_{opt}$ as checkpoint intervals at which we measure the number of I/O operations and the execution time, where τ_{opt} is the optimal checkpoint interval with respect to the execution time. Recall that τ_{opt} depends on system MTTI and since we use two values of system MTTI this leads to 8 experiments in total.

Configuration and parameter values:

- Number of Nodes: 4 nodes (64 cores)
- Specification of Ranger Node: Each node contains four AMD Opteron Quad-Core 64-bit processors (16 cores in all) on a single board, as an SMP unit. The core frequency is 2.3 GHz and supports 4 floating-point operations per clock period with a peak performance of 9.2 GFLOPS/core or 128 GFLOPS/node. Each node contains 32 GB of memory. The memory subsystem has a 1.0 GHz HyperTransport system Bus, and 2 channels with 667 MHz DDR2 DIMMS. Each socket possesses an independent memory controller connected directly to an L3 cache [RangerUserGuide].
- Input data set: Dataset containing 17000 gene sequences (taxa) of 1303 genetic characters
- System MTTI: 6 hours and 8 hours
- Number of checkpoint intervals: 4 per value of MTTI, 8 total.
- Number of experiments: total of 24, three repetitions for each distinct combination of MTTI and checkpoint interval.

Table 4.6: Results from RAxML-Light Execution

MTTI (hrs)	Checkpoint Interval (secs)	Average Execution Time (hrs)	Number of Checkpoint Operations	Rework Time (secs)
6	26 ($\frac{1}{2}\tau_{opt}$)	21.02	2,761	73
	52 (τ_{opt})	21.80	1,467	67
	114 ($2\tau_{opt}$)	21.36	637	189
	228 ($4\tau_{opt}$)	26.05	353	496
8	30 ($\frac{1}{2}\tau_{opt}$)	20.32	2,761	33
	60 (τ_{opt})	21.40	1298	36
	120 ($2\tau_{opt}$)	19.02	637	99
	240 ($4\tau_{opt}$)	19.20	353	586

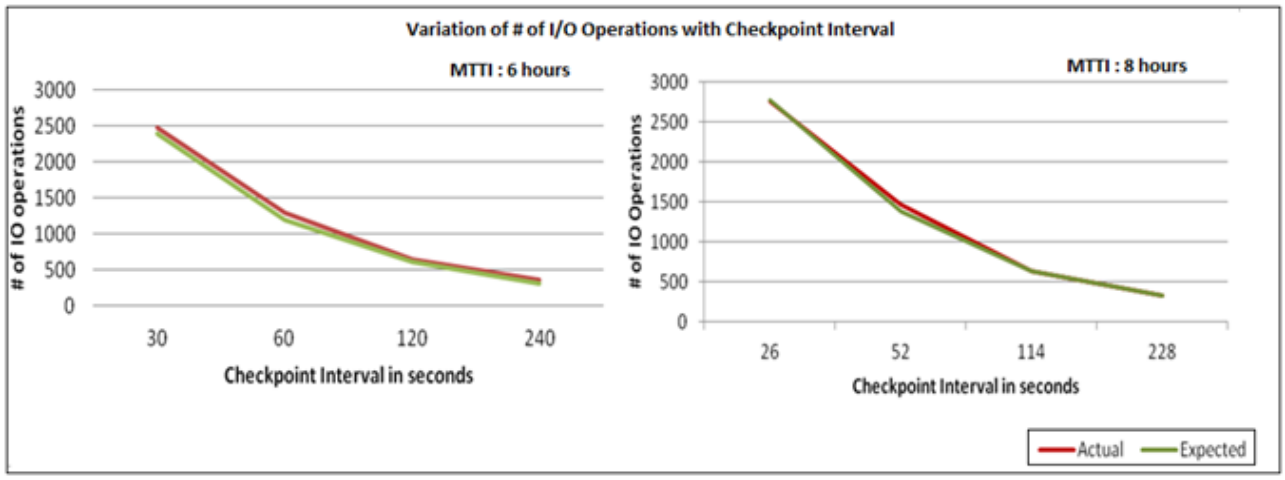
- Solution time: 20 hours

In Table 4.6, we present the checkpoint intervals used, the MTTI used,

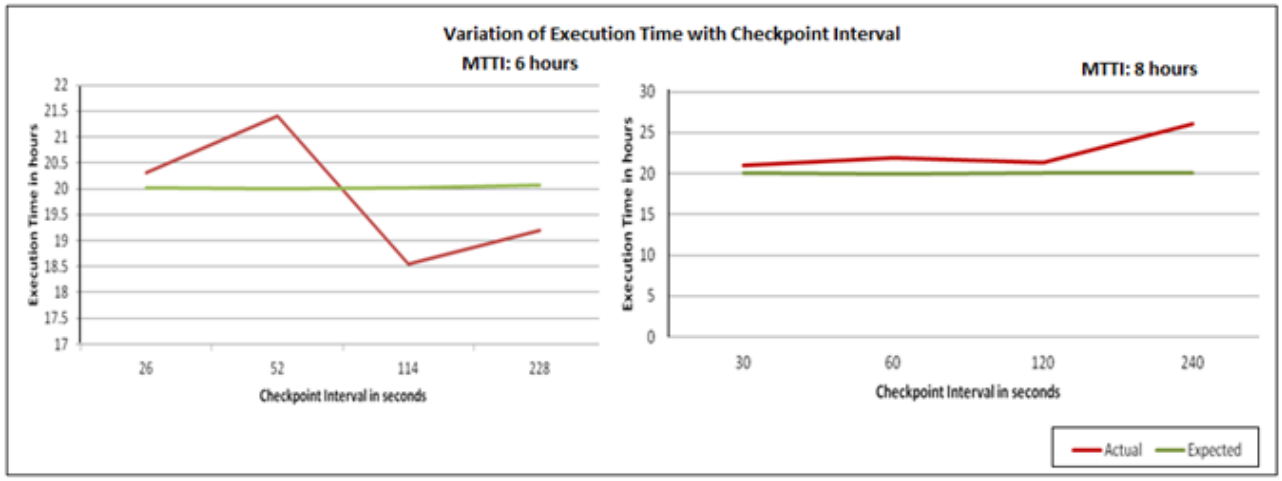
Figure 4.5a depicts the variation in the measured number of I/O operations in terms of the checkpoint interval in relationship to the number of I/O operations estimated by the model described in [12] (green line). As can be seen, the actual number conforms closely to the the values estimated by analytical model; the error is less than 6% for an MTTI of six hours and 15% for an MTTI of eight hours. These results validate the prediction of the analytical model, i.e., that by increasing the checkpoint interval to twice the optimal value with respect to execution time, the number of I/O operations is reduced to almost half.

Figure 4.5b depicts the variation in execution time in terms of the checkpoint interval in relationship to the expected execution time estimated by the model described in [21] (green line). The execution time behavior does not conform to the analytical model. The percentage error varied from 5% to 30% for MTTI of 8 hours and 1.5% to 8% for MTTI of 6 hours. However, as the number of experiments conducted was only 3 and the condition of experiments varied, the anomaly could not be explained.

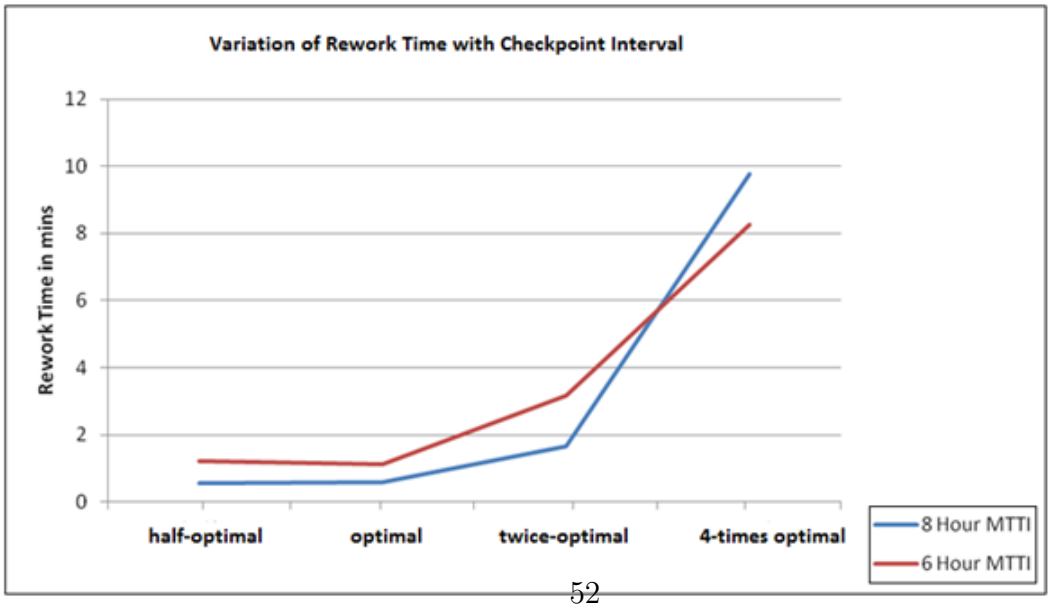
Figure 4.5c depicts the rework time due to failures encountered in any simulation. As expected the rework time should increase with increase in checkpoint interval. The rework



(a) Number of I/O operations vs. checkpoint interval



(b) Execution Time vs. checkpoint interval



(c) Rework Time vs. checkpoint interval

Figure 4.5: RAxML-Light on Ranger with System MTTI 6 and 8 hours

time is similar for optimal checkpoint interval and half optimal checkpoint interval is same. Also for twice optimal the rework time in only 1 - 2 minutes greater than optimal checkpoint interval. Thus, checkpointing at a higher frequency does not cause greater impact on rework time.

The main motivation behind running these simulations was to collect the latency data of running the dataset, mentioned in earlier segments of this section 4.2.2. This experiment was done to understand how checkpoint latency scales with the number of nodes. For this RAxML-Light was run on 16 and 100 nodes of Ranger. In Table 4.7, the latency observed has been presented

Table 4.7: Latency of RAxML-Light dataset running on different number of nodes on Ranger

Sl. No	**Number of Nodes**	**Latency (average of 30 writes) in seconds**
1	16	0.102
2	100	0.17

4.3 Simulator with LANL failure data

Our simulation study (the details of which are described in Section 3.3 uses the checkpoint latency defined by running RAxML-Light on 100 nodes of Ranger (i.e., 0.17 seconds) and MTTIs based on the hardware failure data from the two LANL clusters with the largest number of nodes, i.e., systems 18 and 19. Based on all failure data, the node MTTI for systems 18 and 19 was 472 days; it was 410 days for system 19 based on failure data collected during the first two years of production. We assume 300 jobs and a job length of 10 days. The checkpoint interval is varied from one minute to 1,400 minutes, in increments of 1 minute till under 100 minutes and in increments of 10 minutes there after. For each checkpoint interval the simulator reports the average execution time and average number of I/O operations of the 300 jobs, which is reported below.

First, in Figure 4.6, we compare the increase (in minutes) in execution time with the increase in the size of the checkpoint interval for simulations on system18. The average simulated execution time of the 300 jobs is depicted by the blue line, while that predicted by Daly's model is depicted by the green line. The percentage error of the model in terms of execution time is less than 2% during the 10-day duration, and the maximum difference between the simulated and predicted execution times is 2.33 hours.

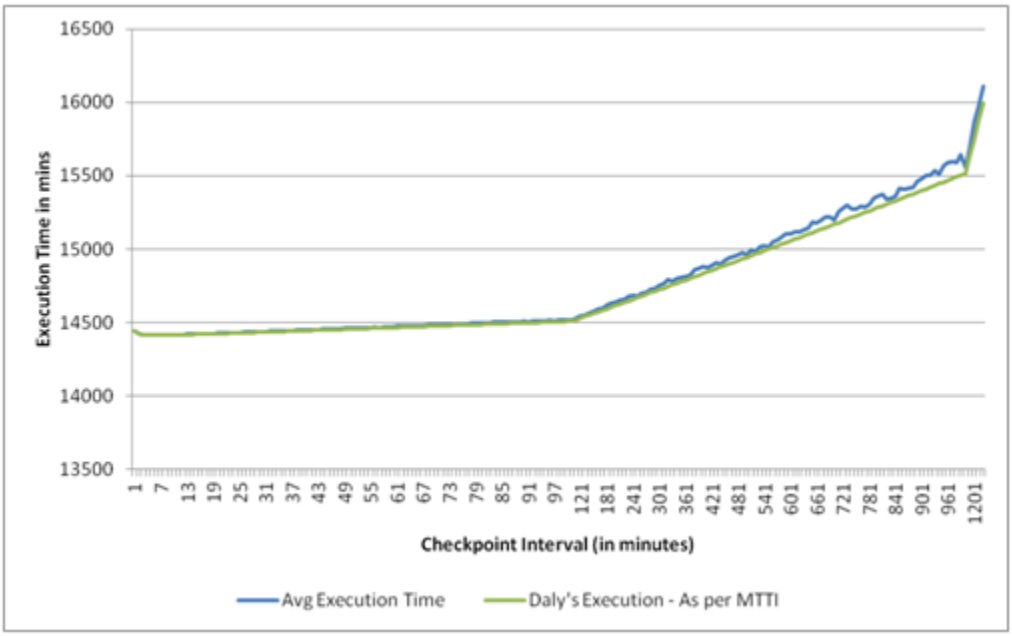


Figure 4.6: Simulation of RAxML-Light with Node MTTI of 472 Days on System 18: Execution Time vs. Checkpoint Interval

Next, Figure 4.7 allows us to compare the simulated number of checkpoint I/O operations with that predicted by Arunagiri's model. The percentage error in the predicted number of checkpoint I/O operations is less than 7%, and the maximum difference between the simulated and predicted number of checkpoint I/O operations is 0.83.

These results show that, even with historic failure data that does not conform to an exponential failure distribution, as is assumed by the models, the checkpoint I/O model and the checkpoint execution-time model both perform well with percentage errors less

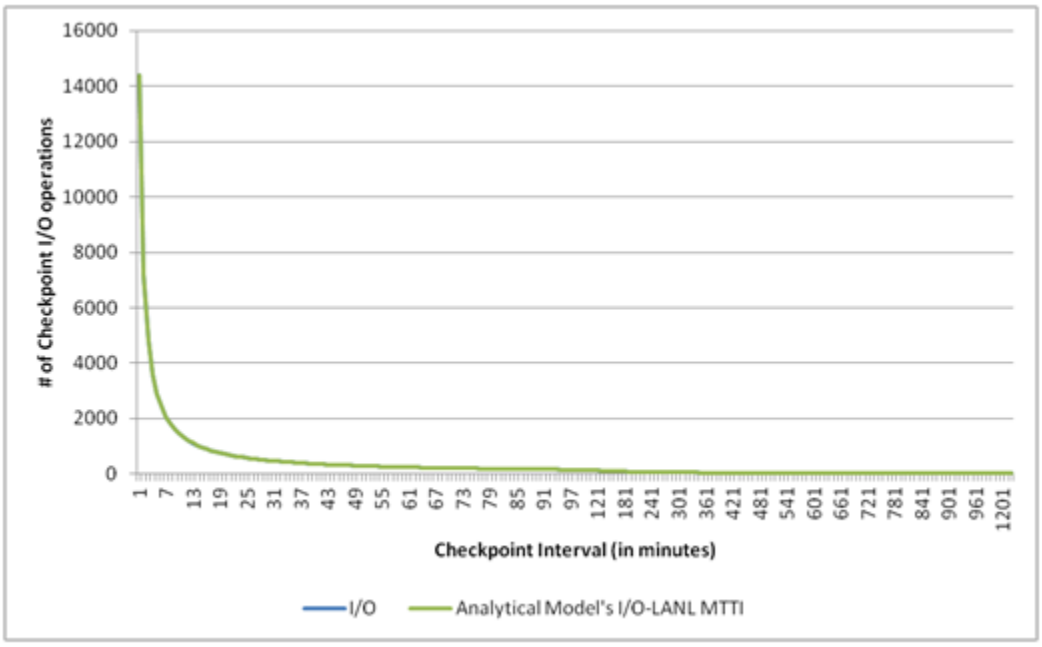


Figure 4.7: Simulation of RAxML-Light with Node MTTI of 472 Days on System 18: Number of Checkpoint I/O Operations vs. Checkpoint Interval

than 7% and 2%, respectively.

To explore whether using a value of MTTI estimated using failure data from the first two years of production can be used in the analytical models with the same effectiveness as the MTTI computed using all the failure data, we performed a similar experiment with two different values of MTTI, i.e., the first computed using all the failure data for system 19, i.e., an MTTI of 472 days, and the second computed using the first two years of this failure data, i.e., an MTTI of 410 days. The results of these experiments follow.

In Figure 4.8, we compare the increase (in minutes) in execution time with the increase in the size of the checkpoint interval for simulations on system19 with MTTI of 411 and 472 days.

In Figure 4.9, we compare the increase (in minutes) in number of I/O operations with the increase in the size of the checkpoint interval for simulations on system19 with MTTI of 411 and 472 days.

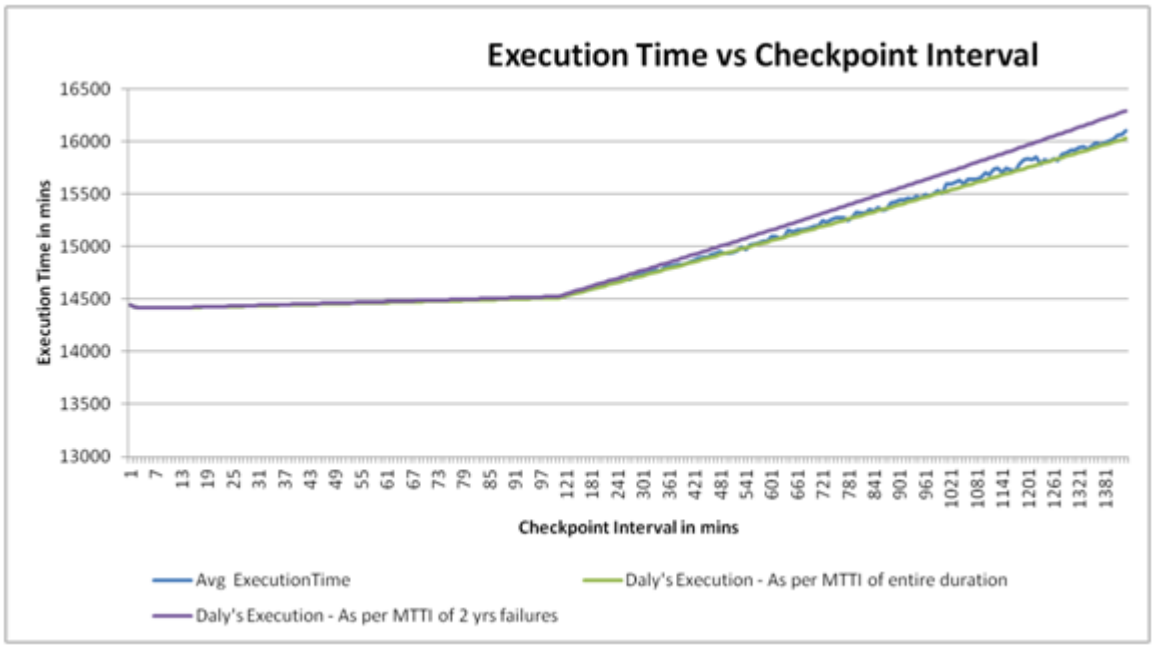


Figure 4.8: Simulation of RAxML-Light with Node MTTI of 472 and 410 Days on System 19: Execution Time vs. Checkpoint Interval

In Figure 4.10, we present the percentage error in using the overall MTTI of 472 days and MTTI based on two years i.e., 410 days.

Figures 4.8 and 4.9 depict the simulated execution time and number of checkpoint I/O operations generated for a range of checkpoint intervals (i.e., from one minute to 1440 minutes, in increments of 1 till less than 100 minutes and increment of 10 above that). As can be seen, Daly's execution-time model tracks the simulated execution time fairly well, with errors increasing as the checkpoint interval increases. In contrast, Arunagiri's model tracks the simulated number of I/O operations very well even for the larger checkpoint intervals.

Figure 4.10 provides a comparison between the results generated by simulations with the MTTI defined by an analysis of all of LANL system 19's failure data (i.e., an MTTI of 472 days) and those with the MTTI defined by an analysis of just the first two years of system 19's failure data (410 days). Using the former, the percentage error of the predicted

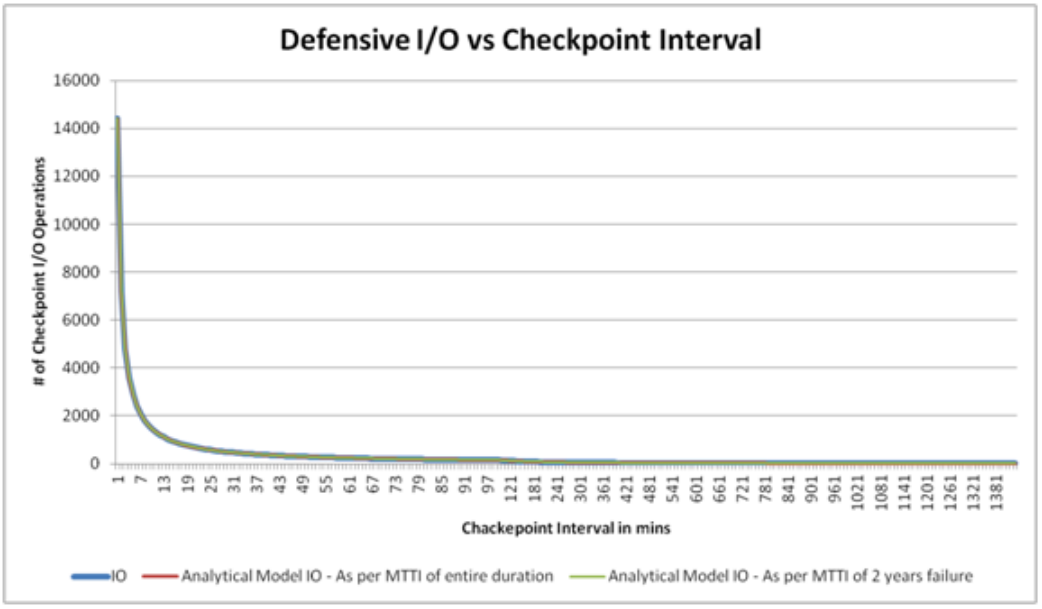


Figure 4.9: Simulation of RAxML-Light with Node MTTI of 472 and 410 Days on System 19: Number of Checkpoint I/O Operations vs. Checkpoint Interval

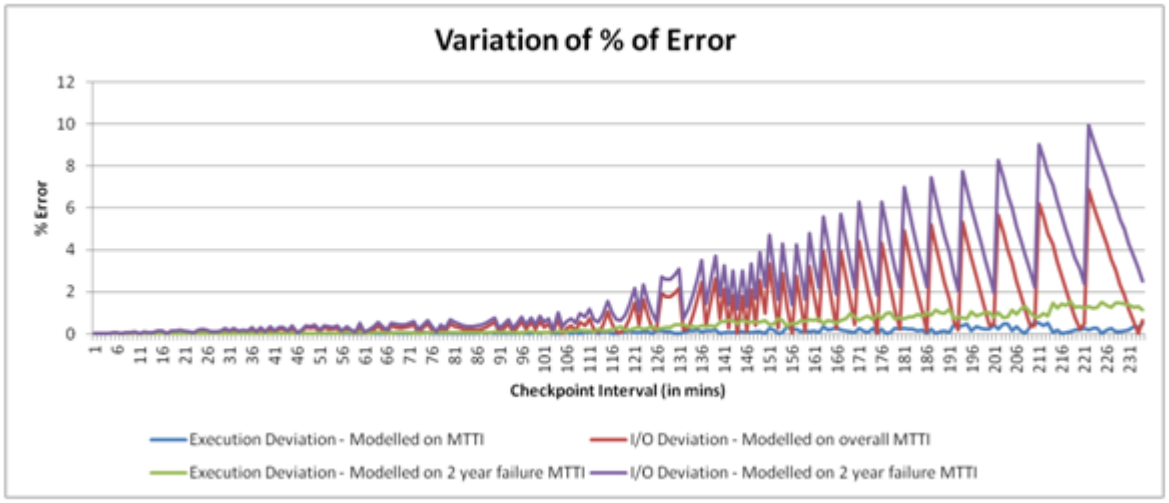


Figure 4.10: Simulation of RAxML-Light with Node MTTI of 472 and 410 Days: Comparison of Execution Times and Number of Checkpoint I/O Operations

execution time is less than 1%, while using the latter; it is less than 2%. In terms of number of checkpoint I/O operations, using the former results in a percentage error of less than 7% and using the former, it is less than 10%. Thus, estimates of wall-clock execution time and the number of checkpoint I/O operations obtained using the analytical models and a value of MTTI computed using failure data from the first two years of production of the system had an error of less than 10% for this system.

Chapter 5

Conclusions and Future Work

Using application runs on a real HPC system and simulations, we showed that for RAxML a popular community code, and workload of 17,000 gene sequences of length 1303 genetic characters, increasing the checkpoint interval to a value that is larger than the optimal checkpoint interval with respect to execution time results in a significant decrease in the number of checkpoint I/O operations without a significant increase in execution time. This shows that the analytical model with respect to checkpoint I/O operations holds good in the studied cases. Our study also shows that for both exponential failure distributions and historic failure data that do not fit an exponential distribution, the analytical model related to checkpoint I/O operations [12] and execution time [21] results in predictions with a maximum error of 10% and 2%, respectively.

Although our study with respect to execution time on Ranger did not yield any conclusive results, the experiments related validating the analytical model for number of checkpoint I/O operations gave positive results with percentage error ranging between 6% and 15%. The runs of RAxML on Ranger also provided the latency parameters of executing RAxML-Light on 100 nodes (1,600 processors). And the failure data for LANL systems 18 and 19 in the CFDR provided an estimate of Node MTTI. In the future we plan to explore the possibility of coordinating the I/O traffic of multiple concurrently executing HPC applications to increase overall HPC system efficiency.

References

[1] Material (test datasets) for 2007 supercomputing paper on parallelizing raxml on the ibm bluegene/l, available: http://wwwkramer.in.tum.de/exelixis/software.html.

[2] Raxml community site at http://sco.h-its.org/exelixis/software.html.

[3] Asci purple statement of work, lawrence livermore national laboratory (http://www.llnl.gov/asci/purple/attachment 02 purplesowv09.pdf), 2006.

[4] The computer failure data repository (cfdr)(http://cfdr.usenix.org/), February 2009.

[5] Dataset - demo_bac_arch.phylip.reduced.bz2 downloaded from iptol website at https://pods.iplantcollaborative.org/wiki/display/iptol/bt_documents, 2010.

[6] Texas advanced computing center (http://www.tacc.utexas.edu/home), 2011.

[7] Big trees project of iplant tree of life collaborative (http://www.iplantcollaborative.org/challenge/iplant-tree-life/big-trees), 2012.

[8] Ranger user guide (http://www.tacc.utexas.edu/user-services/user-guides/ranger-user-guide), 2012.

[9] S. Agarwal, R. Garg, M. S. Gupta, and J. E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on supercomputing*, ICS '04, pages 277–286, New York, NY, USA, 2004. ACM.

[10] T.W. Anderson and D. A. Darling. A test of goodness of fit.

[11] J. Ansel, K. Arya, and G. Cooperman. Dmtcp: Transparent checkpointing for cluster computations and the desktop. In *Proceedings of the 2009 IEEE International Sym-*

posium on Parallel&Distributed Processing, IPDPS '09, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[12] S. Arunagiri, J. T. Daly, and P. J. Teller. Modeling and analysis of checkpoint i/o operations. In *Proceedings of the 16th International Conference on Analytical and Stochastic Modeling Techniques and Applications*, ASMTA '09, pages 386–400, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] B. Bhargava and Shu-Renn Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Proc. of the 7th IEEE Symposium on Reliable Distributed Systems*, 1988.

[14] D. Briatico, A. Ciuffoletti, and L. Simoncini. A distributed domino-effect free recovery algorithm. In *4th IEEE Symposyum on Reliability in Distributed Software and Database Systems*, 1984.

[15] F. Cappello. Fault tolerance in petascale/exascale systems: current knowledge, challenges and research opportunities. *Int. J. High Perform. Comput. Appl.*, 23(3):212–226, August 2009.

[16] P. H. Carns, R. Latham, R. B. Ross, K. Iskra, S. Lang, and K. Riley. 24/7 characterization of petascale i/o workloads. In *CLUSTER*, pages 1–10. IEEE, 2009.

[17] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[18] Y. Chen, X. Sun, R. Thakur, H. Song, and H. Jin. Improving parallel i/o performance with data layout awareness. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, Cluster '10, pages 302–311, Washington, DC, USA, 2010. IEEE Computer Society.

[19] John T. Daly. A model for predicting the optimum checkpoint interval for restart dumps. In *Proceedings of the 2003 international conference on Computational science*, ICCS'03, pages 3–12, Berlin, Heidelberg, 2003. Springer-Verlag.

[20] John T. Daly. A strategy for running large scale applications based on a model that optimizes the checkpoint interval for restart dumps. pages 70–74, Edinburgh, Scotland, UK, May 2004.

[21] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Gener. Comput. Syst.*, 22(3):303–312, February 2006.

[22] D. A. Darling. The annals of mathematical statistics.

[23] J. M. Dennis and R. Loft. Optimizing high-resolution climate variability experiments on the cray xt4 and xt5 systems at nics and nersc. In *Cray User Group (CUG09)*, Atlanta, GA, April 2009.

[24] Elmootazbellah N. E. and James S. P. Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery. *IEEE Transactions on Dependable and Secure Computing*, 1:97–108, 2004.

[25] S. Fu and C. Z. Xu. Exploring event correlation for failure prediction in coalitions of clusters. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 41:1–41:12, New York, NY, USA, 2007. ACM.

[26] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 27–32, New York, NY, USA, 2002. ACM.

[27] A. Geist and C. Engelmann. Development of naturally fault tolerant algorithms for computing on 100,000 processors, 2002.

[28] G. A. Gibson, B. Schroeder, and J. Digney. Failure tolerance in petascale computers, November 2007.

[29] R. Guerraoui and A. Schiper. Fault-tolerance by replication in distributed systems. In *Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies*, Ada-Europe '96, pages 38–57, London, UK, UK, 1996. Springer-Verlag.

[30] T. J. Hacker, F. Romero, and C. D. Carothers. An analysis of clustered failures on large supercomputing systems. *J. Parallel Distrib. Comput.*, 69(7):652–665, July 2009.

[31] Michael Harney. Fault tolerance: Validating checkpoint/restart analytical models using namd (to be published), 2012.

[32] T. Heath, R. P. Martin, and T. D. Nguyen. Improving cluster availability using workstation validation. *SIGMETRICS Perform. Eval. Rev.*, 30(1):217–227, June 2002.

[33] W. M. Jones, J. T. Daly, and N. DeBardeleben. Impact of sub-optimal checkpoint intervals on application efficiency in computational clusters. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 276–279, New York, NY, USA, 2010. ACM.

[34] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The failure trace archive: enabling comparative analysis of failures in diverse distributed systems. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 398–407, Washington, DC, USA, 2010. IEEE Computer Society.

[35] R. Latham, C. Daley, W. Liao, K. Gao, R. Ross, A. Dubey, and A. Choudhary. A case study for scientific i/o: Improving the flash astrophysics code. *The Journal of Computational Science and Discovery*, accepted, 2012.

[36] P. Lemarinier, A. Bouteiller, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. *Int. J. High Perform. Comput. Netw.*, 2(2-4):146–155, February 2004.

[37] Y. Ling, J. Mi, and X. Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Trans. Comput.*, 50(7):699–708, July 2001.

[38] Y. Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and S. Scott. A reliability-aware approach for an optimal checkpoint/restart model in hpc environments. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, Cluster '07, pages 452–457, Washington, DC, USA, 2007. IEEE Computer Society.

[39] J. Lofstead, M. Polte, G. Gibson, S. Klasky, K. Schwan, R. Oldfield, M. Wolf, and Q. Liu. Six degrees of scientific data: reading patterns for extreme scale science IO. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 49–60. ACM, 2011.

[40] D. Manivannan and M. Singhal. A low-overhead recovery technique using quasi-synchronous checkpointing. In *Proc. IEEE Int. Conference on Distributed Computing Systems*, pages 100–107, 1996.

[41] F. J. Massey. The kolmogorov-smirov test for goodness of fit.

[42] Harish Gapanati Naik, Rinku Gupta, and Pete Beckman. Analyzing checkpointing trends for applications on the ibm bluegene/p system. In *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ICPPW '09, pages 81–88, Washington, DC, USA, 2009. IEEE Computer Society.

[43] R. Narasimha, L. Yudan, B. L. Chokchai, N. Raja, and S. Stephen. Reliability analysis in hpc clusters.

[44] T. Narate, N. Nichamon, C. Clayton, E. James, L. Chokchai, O. George, L. S. Stephen, and E. Christian. Bluegene/l log analysis and time to interrupt estimation. *Availability, Reliability and Security, International Conference on*, 0:173–180, 2009.

[45] Ron A. Oldfield, Sarala Arunagiri, Patricia J. Teller, Seetharami Seelam, Maria Ruiz Varela, Rolf Riesen, and Philip C. Roth. Modeling the impact of checkpoints on next-generation systems. In *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, MSST '07, pages 30–46, Washington, DC, USA, 2007. IEEE Computer Society.

[46] Adam J. Oliner, Larry Rudolph, and Ramendra K. Sahoo. Cooperative checkpointing: a robust approach to large-scale systems reliability. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 14–23, New York, NY, USA, 2006. ACM.

[47] X. Ouyang, S. Marcarelli, and D. K. Panda. Enhancing checkpoint performance with staging io and ssd. In *Proceedings of the 2010 International Workshop on Storage Network Architecture and Parallel I/Os*, SNAPI '10, pages 13–20, Washington, DC, USA, 2010. IEEE Computer Society.

[48] K. Pattabiraman, C. Vick, and A. Wood. Modeling coordinated checkpointing for large-scale supercomputers. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 812–821, Washington, DC, USA, 2005. IEEE Computer Society.

[49] A. Rokas. Phylogenetic analysis of protein sequence data using the randomized axelerated maximum likelihood (raxml) program. *Curr Protoc Mol Biol*, Chapter 19:Unit19.11, 2011.

[50] R. K. Sahoo, A. Sivasubramaniam, M. S. Squillante, and Y. Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *Proceedings of the 2004*

International Conference on Dependable Systems and Networks, DSN '04, pages 772–, Washington, DC, USA, 2004. IEEE Computer Society.

[51] M. Schneider. Self-stabilization. *ACM Comput. Surv.*, 25(1):45–67, March 1993.

[52] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. *IEEE Trans. Dependable Secur. Comput.*, 7(4):337–351, October 2010.

[53] A. Stamatakis. Raxml-vi-hpc: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, October 2006.

[54] A. Stamatakis, A. J. Aberer, C. Goll, S.A. Smith, and S.A. Berger. Raxml-light: a tool for computing terabyte phylogenies. *Bioinformatics*, 00:1–2, 2005.

[55] A. Stamatakis and M. Ott. Exploiting fine-grained parallelism in the phylogenetic likelihood function with mpi, pthreads, and openmp: a performance study. In *Proceedings of the Third IAPR International Conference on Pattern Recognition in Bioinformatics*, PRIB '08, pages 424–435, Berlin, Heidelberg, 2008. Springer-Verlag.

[56] Alexandros Stamatakis. The raxml 7.0.4 manual.

[57] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Trans. Comput. Syst.*, 3(3):204–226, August 1985.

[58] ASCAC subcommittee on exascale computing. The opportunities and challenges of exascale computing, 2010.

[59] R. Subramaniyan, E. Grobelny, S. Studham, and A. D. George. Optimization of checkpointing-related i/o for high-performance parallel and distributed computing. *J. Supercomput.*, 46(2):150–180, November 2008.

[60] T. Tantikul and D. Manivannan. A communication-induced checkpointing and asynchronous recovery protocol for mobile computing systems. In *Proceedings of the Sixth International Conference on Parallel and Distributed Computing Applications and*

Technologies, PDCAT '05, pages 70–74, Washington, DC, USA, 2005. IEEE Computer Society.

[61] T. Thanakornworakij, R. Nassar, Chokchai B. Leangsuksun, and M. Paun. The effect of correlated failure on the reliability of hpc systems. In *Proceedings of the 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications Workshops*, ISPAW '11, pages 284–288, Washington, DC, USA, 2011. IEEE Computer Society.

[62] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE Trans. Comput.*, 46(8):942–947, August 1997.

[63] P. Wang, Y. Du, H. Fu, H. Zhou, X. Yang, and W. Yang. A novel fault-tolerant parallel algorithm. In *Proceedings of the 7th international conference on Advanced parallel processing technologies*, APPT'07, pages 18–29, Berlin, Heidelberg, 2007. Springer-Verlag.

[64] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked windows nt system field failure data analysis. In *Proceedings of the 1999 Pacific Rim International Symposium on Dependable Computing*, PRDC '99, pages 178–, Washington, DC, USA, 1999. IEEE Computer Society.

[65] J.W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, September 1974.

[66] G. Zarza, D. Lugones, F. Daniel, and E. Luque. Fault-tolerant routing for multiple permanent and non-permanent faults in hpc systems. In Hamid R. Arabnia, Steve C. Chiu, George A. Gravvanis, Minoru Ito, Kazuki Joe, Hiroaki Nishikawa, and Ashu M. G. Solo, editors, *PDPTA*, pages 144–150. CSREA Press, 2010.

[67] G. Zheng, L. Shi, and L. V. Kale. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *Proceedings of the 2004 IEEE International*

Conference on Cluster Computing, CLUSTER '04, pages 93–103, Washington, DC, USA, 2004. IEEE Computer Society.

[68] Z. Zheng, L. Yu, W. Tang, Z. Lan, R. Gupta, N. Desai, S. Coghlan, and D. Buettner. Co-analysis of ras log and job log on bluegene/p. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 840–851, Washington, DC, USA, 2011. IEEE Computer Society.

Appendix A

Code Modifications

A.1 RAxML-Light Modified Code

```
#ifdef _DOEPROJ
static int failureArr[MAXVAL];
static double initialTime;
static double nextfailuretime;
static double cntfailures;
static int initial=0;
void invokeCheckpoint(tree *tr){
  checkPointState ckp1;
  double delta, t = gettime(), start=0, end=0;
  char restartFile[2048], buf[64];
  int data,w,i=0;
  FILE* mttiFile;
  FILE *f;
  if(initial==0){
        //Comment: Reading Failure file first time.
        initialTime = t;
        mttiFile=fopen(mtti_file_name,``r'');
        if (mttiFile!=0)
        {
          while((data = fscanf(mttiFile,``%d'', &w)) != EOF)
```

```
        {
                if(data){
                failureArr[i]=w;
                i++;
              }
        }
        nextfailuretime=(double)failureArr[0];
        cntfailures=i;
        fclose(mttiFile);
      }
      initial = 1;
      checkpointsize=0;
}
//Comment: This is the latest checkpoint file to be used for restart.
strcpy(restartFile, binaryCheckpointName);
strcat(restartFile, ''_'');
sprintf(buf, ''%d'', ckpCount);
strcat(restartFile, buf);
//Comment: Time for failure to occur
if((initial< cntfailures) &&
      ((delta = (t - initialTime)) > nextfailuretime))
{
      if(chkFlag==1){
        restartsize=0;
        strcpy(binaryCheckpointInputName,restartFile);
        start=gettime();
        simRestart(tr,initadef);
        end=gettime();
```

```
        logTime(``Restart Start Time = '' + start + ``, ''
                + ``End Time = '' + end +  ``, ''
                + ``Duration = '' + (end-start) + ``, ''
                + ``Size of data write = ''+ restartsize +``(Bytes)'');


        }else{
          printf(``Info: No restart File start from beginning....\n'');
        }
        initial++;
        nextfailuretime = failureArr[initial-1];
    }
    else  if((delta = (t - lastCheckpointTime)) > checkpointInterval)
    {
        //Comment: Writing Checkpoint after checkpoint interval
        lastCheckpointTime = t;
        start=gettime();
        //Comment: trCkpState--> Saved after completion of SPR cycle
        writeCheckpoint(trCkpState);
        end=gettime();
        logTime(''Checkpoint Start Time = '' + start + ``, ''
                + ``End Time = '' + end +  ``, ''
                + ``Duration = '' + (end-start) + ``, ''
                + ``Size of data write = ''+  checkpointsize +``(Bytes)'');
        checkpointsize=0;
        }
}
#endif
```

A.2 Code for Curve Fitting

```
library(fitdistrplus)

##Change system as per requirement
system=2
dataFile <- read.csv(``data.csv'',head=TRUE,sep=``,'')
newFile <- na.omit(dataFile) # ignore NA values
filterFile <- subset(newFile,newFile$system==i)
# change here to select any other column
Y <- c(filterFile$last.failure)
fcount <- length(Y)

##Fit Weibull
M <- fitdist(Y,``weibull'')
print(M)
plot(M)
gofstat(M,print.test=``TRUE'')

##Fit Exponential
M1 <- fitdist(Y,``exp'')
print(M1)
plot(M1)
gofstat(M1,print.test=``TRUE'')
```

Curriculum Vitae

Bidisha Chakraborty was born on February 29, 1984. The youngest daughter of Amareswar Chakraborty and Joyashree Chakraborty, she graduated from Frank Anthony Public School, Kolkata, India in the spring of 2003. She entered Institute of Engineering and Management in the fall of 2003, and, received her Bachelors in Information Technology in spring of 2007. In 2007 she joined Tata Consultancy Services where she served for three years as Assistant Systems Engineer. In January 2010, she married Aritra DattaGupta.

In the fall of 2010, she entered the Graduate School of The University of Texas at El Paso. While pursuing a master's degree in Computer Science she worked as a Research Assistant in the research lab of Drs. Patricia J. Teller and Sarala Arunagiri (HiPerSys).

Permanent address: 1700 Hawthorne Street, Apt 302
El Paso, Texas 79902