2013-01-01

A Case Study Towards Verification of the Utility of Analytical Models in Selecting Checkpoint Intervals

Michael Joseph Harney
University of Texas at El Paso, mjharney@miners.utep.edu

A CASE STUDY TOWARDS THE VERIFICATION OF THE UTILITY OF ANALYTICAL MODELS IN SELECTING CHECKPOINT INTERVALS

MICHAEL JOSEPH HARNEY

Department of Computer Science

Approved :

_____
Patricia Teller, Ph.D., Chair

_____
Sarala Arunagiri, Ph.D.

_____
Michael McGarry, Ph.D.

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

A CASE STUDY TOWARDS THE VERIFICATION OF THE UTILITY OF ANALYTICAL MODELS IN SELECTING CHECKPOINT INTERVALS

By

MICHAEL JOSEPH HARNEY, B.S.

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 Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

May 2013

Table of Contents

List of Figures

List of Tables

Chapter 1

1.1 Introduction

Problems such as biomolecular simulation, astrophysics simulation, and geological surveying require more computing resources to obtain better, more detailed information and insights about simulated interactions. Such computations run on a single computer can be prohibitive in terms of the computation time needed. Because of this, it becomes necessary to use parallel computation. By distributing the work across 16, 64, 256, and even greater numbers of computer nodes one can complete such computations in much less time. As the size, in terms of the number of computer nodes, of a High Performance Computing (HPC) system grows, the size of the problems that can be solved can grow.

While this execution-time speed-up is desirable, HPC systems come with their own challenges. One of the key challenges is system reliability. All computer components have an expected time to failure. Although the actual time at which a component will fail is unknown, a statistical estimate of the expected time between failures, called Mean Time Between Failures (MTBF) or the Mean Time To Interrupt (MTTI), can be computed. The MTBF of a single computer node can easily be more than one year, so this does not seem to be a large concern. However, when a job is executed by hundreds or even thousands of nodes the MTTI of the whole system on which the job is run can be as small as a few hours. Therefore, to ensure the completion of a job that has a *solution time* of many hours or days, which is the time taken to

execute the job to completion in a failure-free environment, it is important to employ a fault tolerance method.

Checkpointing is a popular fault tolerance method in which the data required to recreate a job's system state is stored at certain intervals of time. In the event of a failure, the computation is resumed from the last saved state, or checkpoint, rather than restarting the job from the beginning. The data stored at each checkpoint operation is called the *checkpoint data* and the time taken to store the data is called the *checkpoint latency*. The time interval between checkpoints may be regular in which case the checkpoint operations are called *periodic checkpoints*. There are a variety of ways that checkpointing can be implemented; these are based on where and how the checkpoint data is stored and how checkpoint intervals are determined. A classification of checkpoint methods is described in Section 1.2.

This thesis focuses on periodic checkpointing where checkpoint data is stored in persistent peripheral memory and, thus, requires an I/O operation, which is expensive in terms of both execution time and utilization of network and I/O resources. In this case, the total number of checkpoint I/O operations and the wall clock execution time of an application depend on the chosen checkpoint interval. Choosing a checkpoint interval that minimizes both the wall clock execution time and the number of checkpoint I/O operations is a challenge. Given a checkpoint interval, analytical models can determine the expected application wall clock execution time and the expected number of checkpoint I/O operations. There are several analytical models that can be used to estimate wall clock execution time based on the assumed failure distribution. In this

thesis we use models based on an exponential failure distribution, which are presented in Section 1.3.

For a popular molecular dynamics community code, NAMD (Not Another Molecular Dynamics) (Philips, Zheng, & Kale, 2002), executed on the Ranger HPC system at Texas Advanced Computing Center (TACC), this thesis investigates the behavior of wall clock execution time and number of I/O operations as a function of the checkpoint interval. We compare this behavior to the trends predicted by the analytical models under consideration thereby investigating the validity of the models. As described in Section 1.5, this study was performed by injecting simulated failures.

While checkpointing provides a method of recovering from failures, it does not address the problem of failure prediction. Researchers in this area have various opinions about the occurrence of failures in HPC systems. Some believe that failures follow an exponential distribution; some assume that failures have a pattern described by a Weibull distribution (Schroeder & Gibson, 2006); others presume that there is a lognormal distribution; or there is no good pattern that describes the occurrence of failures (Oliner & Stearley, 2007). There also are questions about whether failures can be treated as independent events. In this thesis we investigate this issue using clustering. We examine failures from a publicly-available failure log and check for predominance of clusters, which may point to failures not being independent events. Note that this is not a proof of dependence. The clustering algorithms examined in this context are presented in Section 1.4.

In summary, Sections 1.2, 1.3, and 1.4 of this chapter provide the background for understanding checkpoint/restart, the analytical models studied, and the clustering algorithms used to study failure data to determine if failures are dependent or independent. Finally, Section 1.5 states the problem addressed by this thesis, enumerates its contributions, and presents the organization of the remainder of the thesis.

1.2 Methods of Checkpointing

Since fault tolerance is an important issue in large-scale HPC, it has been worked on extensively and, thus, many different checkpoint methods exist. These methods can be classified according to where checkpoint data is stored, how it is stored, and when a checkpoint operation is executed. Below we present the choices associated with each of these characteristics, along with related benefits and drawbacks.

1.2.1 Where Checkpoint Data is Stored

Checkpoint data can be stored in memory, on a local hard disk, or in a network file system. In addition, the checkpoint data of each node can be stored in a separate file or the checkpoint data of all the nodes participating in the execution of an application can be stored in a single file. Checkpoint methods that employ these various strategies are described below.

In-memory Checkpointing: In checkpoint methods that use in-memory checkpointing each node associated with a job writes a checkpoint file, which is sent to one or more nodes in the system working on the same job. In the event of a single node failure, this allows for quick

restoration of state data from the lost node. The drawback is that if multiple nodes fail before recovery then there is a chance that the required checkpoint data may be lost and unrecoverable. Moreover, this method requires additional memory on the nodes to store checkpoints and, therefore, it may not be possible to use this method for computations where large amounts of checkpoint data are generated (Moody, 2009).

Local-disk Checkpointing: Checkpoint methods that utilize local-disk checkpointing are similar to those that employ in-memory checkpointing in that each node associated with a job shares checkpoint data with another node running the job. However, the difference is that rather than storing the data in memory, the data is stored in local hard disk. This is slower than in-memory checkpointing because memory is faster to access than disks. However, in contrast to in-memory checkpointing, local-disk checkpointing ensures that the required checkpoint data is recoverable even when multiple failures occur before recovery, but the same nodes must be used to resume the job since the checkpoint data is stored on the local hard disks of these nodes (Moody, 2009). Another critical drawback of this scheme is that some HPC centers, such as Texas Advanced Computing Center (TACC), do not allow programs to write to local hard disks, only RAM disks that are cleared when the job terminates are available locally, thus, making local-disk checkpointing unusable at such centers (Texas Advanced Computing Center Ranger-User-Guide).

Network File System Node Checkpoints: Employing network file system node checkpoints is much the same as using local-disk checkpointing but rather than storing checkpoint data locally or with a neighbor, a node stores it on a network file system. The benefit is that the restart (checkpoint) files of all the nodes cooperating in the execution of the job are stored on the network file system. This allows a failed node to be replaced with a new node

provided that that new node has the capability to run what the failed node was executing. Depending on the checkpoint data stored, the new node may need to have similar hardware for system dependent checkpoints or just need to be able to execute the same program. The drawback of this is that writing to the network file system takes longer than writing to a local hard disk. Additionally, the job must continue to run on the same number of nodes; the job cannot be completed on more or fewer nodes (Ansel, Arya, & Cooperman, 2009).

Single Checkpoint File on a Network File System: Using a single checkpoint file on a network file system, a single node collects all the checkpoint data and writes the entire checkpoint to a single network file. The benefit of this is that, after a restart, the job may be able to run on a different number or set of nodes than that on which it ran before the failure if the specific checkpoint/restart method used by the program allows for this. The drawback is that, compared to the aforementioned ways of storing checkpoint data, this technique takes the greatest amount of time to execute and the execution time grows superlinearly (Moody, 2009).

1.2.2 How Checkpoint Data is Stored

The classification described in Section 1.2.1 is based only on storage options. Checkpointing may seem simple, but when it is run across multiple nodes, it is important to run the checkpoints when it is "safe". If communications, files, and other resources are not handled carefully, the checkpoint may not record data from important events or communications between nodes, losing data needed for proper job restart. Because of this when multiple nodes are executing a job, checkpoints are executed either synchronously or asynchronously in the ways described below.

Synchronous Checkpointing: When checkpoint data is stored synchronously, all running processes are stopped at the point when all are at a safe, uniform state so that checkpointing can be performed without loss of network communication data or other in-flight data. This technique avoids potential problems by ensuring that all file writes have finished and all network traffic has cleared before the checkpoint takes place. The downside of this technique is that the checkpoint latency can become very large as processes must synchronize and wait for network traffic and file writes to clear. Synchronous checkpointing does not scale well because the more nodes involved in the execution of a job, the more time is needed to synchronize process network communications and access to file resources. This makes synchronous checkpointing undesirable for checkpoints run across a large number of nodes (Ansel, Arya, & Cooperman, 2009).

Asynchronous Checkpointing: Although asynchronous checkpointing does not incur the higher overhead of synchronous checkpointing, it has the added complexity of having to ensure that all the processes are in a consistent state when they checkpoint. In certain cases, like a physical simulation where the checkpoint only needs to store the coordinates of objects at a time step, this can be trivial; however, in cases where data from network communications or file reads and writes need to be considered, this adds additional complexity to the checkpointing algorithm.

The research presented in this thesis uses the molecular dynamics program NAMD, which utilizes asynchronous, network file system checkpointing. As a result, upon a failure, NAMD can be restarted from a checkpoint on any number of nodes and is not hardware specific. To restart the job from a checkpoint, the nodes need only the checkpoint data and a compatible

version of NAMD. This technique provides a reliable form of checkpointing but it does not scale linearly.

1.2.3 When a Checkpoint Operation is Executed

A good balance is needed to reach an optimal amount of checkpointing to prevent losses while simultaneously preventing consumption of resources due to checkpointing. To this end many techniques have been examined for determining the best times at which to checkpoint; the main techniques are described below.

Risk-based Checkpointing: This technique determines the times to checkpoint using information about system health, probability, and algorithms to determine the likelihood of a failure occurring in the near future. If the algorithm determines that a failure is likely to occur, the program is instructed to checkpoint (Zheng, Lan, Park, & Geist, 2009).

Interval-based Periodic Checkpointing: Interval-based periodic checkpointing checkpoints at fixed intervals, either in time or in program steps. Many mathematical models exist that predict the optimal checkpoint interval to reduce execution time or the number of I/O operations. (Arunagiri, Daly, & Teller, 2009) (Daly, 2006) (Liu, et al., 2007).

Exponential Back-off Checkpointing: This technique decreases or increases the interval between checkpoints based on predictions made from the observed frequency of failure events (Liu, et al., 2007).

This research primarily focuses on NAMD, which uses an interval-based periodic form of checkpointing. The reason this technique was chosen is that it is the easiest to implement at a HPC center and has been demonstrated to outperform both risk-based and exponential back-off checkpointing methods (Liu, et al., 2007).

1.3 Analytical Models

As discussed, checkpointing is vital to preventing the loss of execution time and resource utilization, but excessive checkpointing leads to the consumption of additional resources. For periodic checkpointing, it is necessary to determine and use the checkpoint interval that enables a job to achieve its target execution time or number of checkpoint intervals, which translates to the number of defensive I/O operations. In this section we describe two analytical models that assume an exponential failure distribution. Given the checkpoint interval, one model estimates the application wall clock execution time and the other estimates the number of checkpoint I/O operations generated.

1.3.1 Model for Application Wall Clock Execution Time:

The first model, due to John Daly (Daly, 2006), enables one to predict the expected wall clock execution time for an application that performs periodic checkpointing. The model is based on multiple factors including solution time, i.e., execution time without checkpoints or failures; checkpoint interval; system MTTI; checkpoint latency; and restart time. As shown in Figure 1, it also allows the prediction of the optimal checkpoint interval for achieving minimum execution time.

$$T = Me^{R/M}\left(e^{(\tau+\delta)/M} - 1\right)\frac{T_s}{\tau} \quad \delta << T_s,$$

FIGURE 1.1 DALY'S EXPECTED WALL CLOCK EXECUTION TIME FORMULA FOR A PERIODIC CHECKPOINTING PROGRAM

where:

T is the expected wall clock execution time, i.e., the total time that the program is expected to execute with checkpoints and failures);

M is the Mean Time To Interrupt (MTTI), i.e., the average expected time between failures for a collection of nodes executing the program;

R is the time for a process to restart after a failure;

δ is the checkpoint latency, i.e., the time spent by the program performing a checkpoint operation;

t is the checkpoint interval, i.e., the interval of time between checkpoint operations;; and

T_s is the solution time, i.e., the execution time of the program if no no checkpoints or failures occurred.

$$\tilde{\tau}_{\text{opt}} = \begin{cases} \sqrt{2\delta M}\left[1 + \frac{1}{3}\left(\frac{\delta}{2M}\right)^{1/2} + \frac{1}{9}\left(\frac{\delta}{2M}\right)\right] - \delta & \text{for } \delta < 2M, \\ M & \text{for } \delta \geq 2M. \end{cases}$$

Figure 1.2 Daly's Estimate for the Checkpoint Interval to Minimize Execution Time.

Mathematical analysis of the model showed that the approximation formula used by Daly always slightly underestimates the optimal checkpoint interval [8]. By computing the difference between the actual optimal checkpoint interval and the estimated optimal checkpoint interval, one could instead overestimate the checkpoint interval by the same margin without incurring

significant increase in execution time, while potentially incurring a significant reduction in the number of defensive I/O operations.

1.3.2 Model for Number of Checkpoint I/O Operations:

The model of Arunagiri, et al., (Arunagiri, Daly, & Teller, 2009) for estimating the number of checkpoint I/O operations is based on the execution time model presented above, and the parameters are the same. With this model, shown in Figure 1.3, one can predict the expected number of defensive I/O operations (consisting of both checkpoint data writes and restart reads) that a job will perform during its course of execution. Using this model, the checkpoint interval that is expected to compute the minimum number of defensive I/O operations can be derived using the formula in Figure 1.4.

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

FIGURE 1.3 FORMULA OF ARUNAGIRI, ET AL. FOR THE EXPECTED NUMBER OF CHECKPOINT I/O OPERATIONS

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

FIGURE 1.4 FORMULA OF ARUNAGIRI, ET AL. FOR THE CHECKPOINT INTERVAL TO MINIMIZE THE NUMBER OF CHECKPOINT I/O OPERATIONS.

Using these models Arunagiri, Teller, and Daly suggest a possible method for reducing checkpoint I/O without incurring significant additional execution time. The graph in Figure 1.6

illustrates this point. This figure shows that, for the area near the optimal checkpoint interval, the checkpoint interval can be increased without dramatically impacting the wall clock execution time (upper curve). Note that this change offers a significant reduction in the number of checkpoint I/O operations (lower curve).

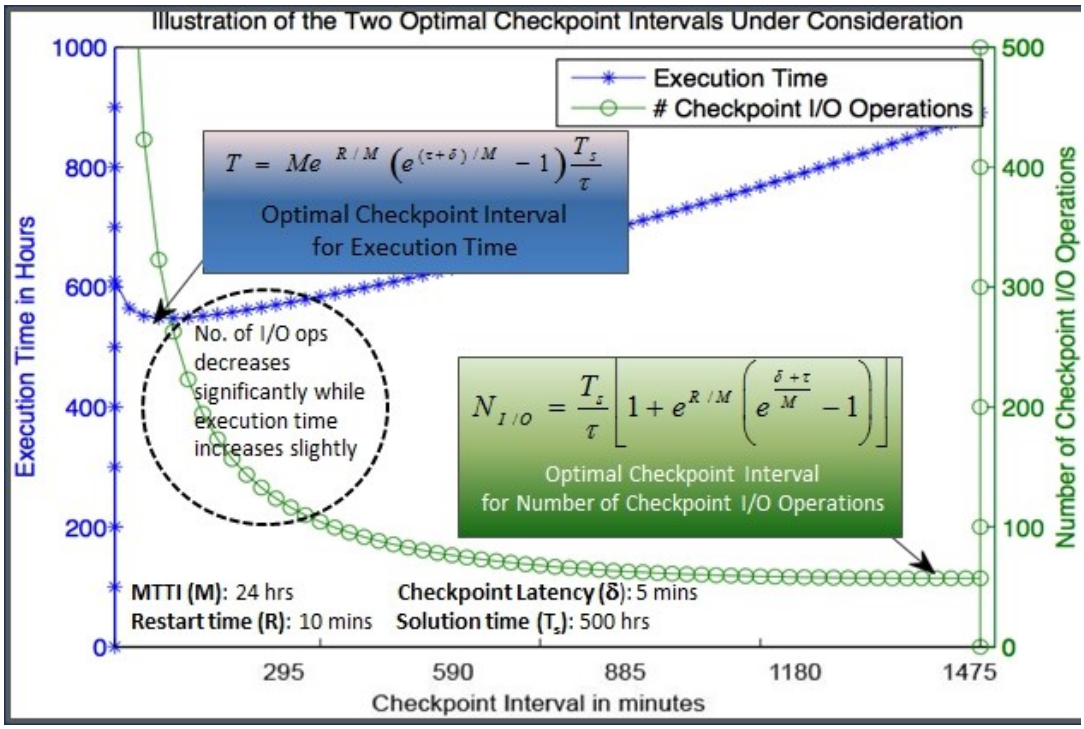

Figure 1.5 Comparison of Execution Time and Number of Checkpoint I/O Operations for Different Checkpoint Intervals.

There are other mathematical models that can be used to predict the same or similar information (Liu, et al., 2007) (Shastry & Venkatesh, 2010), but each is based on sometimes unknown and/or potentially variable parameters, and all are based on certain assumptions. Some of these parameters are: system MTTI, checkpoint interval, checkpoint latency, restart time, and solution time.

The models described above assume an exponential failure distribution. This is a questionable assumption as many researchers studying failure distributions have shown that failures show closer correlation to a Wiebull or log normal distribution (Oliner & Stearley, 2007), and some researchers argue that failures are totally unpredictable and do not fit any distribution. These models also are based on the assumption that all the parameters (MTTI, checkpoint latency, etc.) remain fairly constant through the execution of a job. This potentially limits the usefulness of these models.

The major goal of this study is to use data collected from executing NAMD on the Ranger HPC system at TACC to investigate how closely the empirical and simulated values of wall clock execution time and number of defensive I/O operations track the values estimated by the analytical models presented above. This information is expected to demonstrate the limitations or usefulness of the analytical models.

1.4 Clustering for Determining the Relatedness of Failure Events.

This may seem like a deviation, but many of the analytical models used to determine the best time to checkpoint make the assumption that failure events are independent. If some events are dependent, this makes that assumption faulty and can render the results of models and algorithms based on the assumption of independence less reliable. Because of this, determining if there are related failure events and the degree to which these events are related becomes important in testing those assumptions. Determining relatedness of failure events is a complex problem, which is often difficult to solve using clustering because there is a small probability that even random events cluster together. However, if there is a predominance of clusters then it could be an indication that not all failures are independent events.

There are many different methods of event clustering, this research focuses on the following three common methods.

Expectation Maximization: Expectation Maximization (E-M) clustering, an example of which is depicted in Figure 1.6, uses a recursive algorithm to fit data points into Gaussian distributions (Gupta & Chen, 2010). It is most useful in detecting clear patterns in failures, like failures affecting multiple nodes or happening more frequently on some nodes.

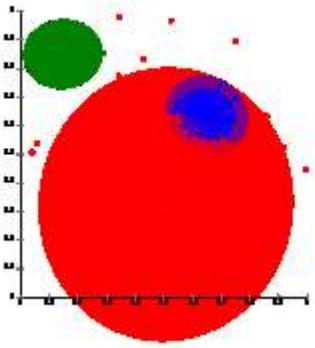

Figure 1.6 Example of E-M Clustering.

K-Means: K-Means clustering, an example of which is presented in Figure 1.7, generates a distribution of a specified number of mean values. It then assigns data points to the closest mean. Subsequently, the means are recalculated based on the number of assigned points. The algorithm repeats until no points are reassigned or until a fixed maximum is reached (K-Means Clustering - Wikipedia). This method is good for detecting smaller clusters of events.

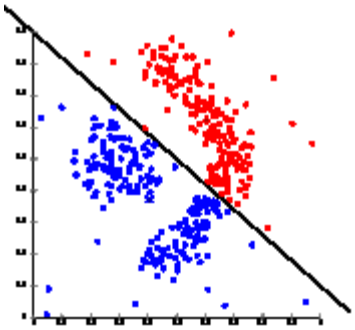

Figure 1.7 Example of K-Means Clustering.

Density-based: Density-based clustering assigns data points to clusters based on how close or far a point is from other points within the cluster (Cluster Analysis - Wikipedia). As shown in the example pictured in Figure 1.8, this method is good for recognizing clusters in which two clearly related groups of events have mean values that would otherwise reassign those points to the wrong cluster.

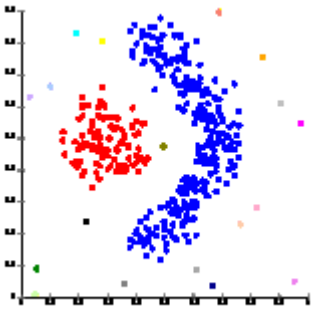

Figure 1.8 Example of Density-based Clustering.

The study presented in this thesis uses E-M and K-Means clustering.

1.5. Problem Definition, Contributions, and Thesis Organization

Ideally, checkpointing should be performed often enough to minimize recomputation. The less recomputation, the smaller the execution time. However, checkpointing, itself, consumes execution time (compute node utilization) and network bandwidth since checkpoint data must be stored to some storage medium, e.g., a network file system. The problem of too infrequent checkpointing is obvious: the larger the checkpoint interval, the larger the expected average amount of execution time lost due to failures. The problem of excessive checkpointing is based on the fact that checkpointing requires execution time and network resources and, thus, decreases application performance. Moreover, too frequent checkpointing can cause I/O contention that can decrease I/O performance and increase application execution time. Because of this, it is important to carefully choose when and how often to checkpoint.

Towards this goal we need to understand how the application wall clock execution time and the number of checkpoint I/O operations generated depend on the checkpoint interval. This thesis makes four major contributions towards enhancing the understanding of the impact of the choice of checkpoint interval by:

(1) Collection of empirical data associated with the execution of NAMD, a popular molecular dynamics code, executed on the Ranger HPC system at TACC that was used to provide parameter values for the models and to validate the models: The empirical data presented includes checkpoint latency, wall clock execution time, and number of checkpoint I/O operations for different values of checkpoint interval. Note that failures were simulated for these runs because large-scale, long-term testing was not a practical

option for quantifying checkpoint overhead for real failures. In addition, an artificially low MTTI was used for the initial testing.

(2) Validation of the analytical models for wall clock execution time and number of checkpoint I/O operations, presented in Section 1.3, using empirical data from NAMD: As mentioned earlier, analytical models are based on several assumptions, e.g., constancy of checkpoint latency, which makes it important to check the accuracy of the model against real data. Data collected from runs of NAMD were used to examine if the analytical models were good predictors even when the assumptions on which the models are based may not be entirely true. This examination showed that checkpoint I/O operations closely followed the expectations set by the analytical model while execution time did not explicitly follow the analytical model.

(3) Collection of simulation data of large-scale NAMD executions using empirical checkpoint latency data and use of this data to determine the accuracy of the analytical models given real failure data: Since the empirical data collected from checkpointing NAMD runs on Ranger used simulated failures, it was important to check the effect that real failure data had on checkpoint overhead. Towards this objective, real checkpoint latency data from NAMD running on Ranger and LANL failure data available at the CFDR (Cluster Failure Data Repository) were used with an event-driven checkpoint simulator to determine the impact that actual failure data would have on checkpoint I/O operations and overhead. The simulated runs showed execution time trends closely matching the analytical model, and the checkpoint I/O operations varying by an average of less than 1 I/O operation on all the data points collected.

(4) Analysis of publicly available failure logs to identify clustered patterns: This was done using clustering to determine what clusters presented the best fit for the LANL failure data based on log normal fitness scores. Failure data from LANL was used to run E-M and K-means clustering. A machine learning program called WEKA was used to analyze the failure data and determine if there might be patterns of related failures in the LANL data. Cluster analysis revealed clear clusters on system 2 and 20 of LANL using EM Clustering, with potential spatio-temporal clustering for all the systems using k-means.

The remainder of this thesis is organized as follows. Chapter 2 presents related research; it describes work that has been done in modeling checkpoints, assumptions made in those models, and research on failure clustering for predictive algorithms. Chapter 3 describes the approaches, tools, and code used to investigate:

- Application checkpoint latency, program execution time, and checkpoint/restart overhead given different runs on a large-scale HPC system;
- Simulation of the execution and failure of jobs using data collected from experiments on a large-scale HPC system plus failure data collected from HPC systems; and
- Cluster analysis of failure data logs of HPC systems to locate potential failure clusters and, thus, investigate failure event interdependence to examine the possibility of checkpointing when clustered events indicate an increased likelihood of failures.

Most importantly, Chapter 4 describes the related experiments and presents experimental results. Finally, Chapter 5 conclusions and future work.

2. Related Work

This chapter presents related work on the following research topics relevant to this study: analytical models for determining the checkpoint interval (i.e., when to checkpoint), HPC failure distribution analysis, and failure clustering. Section 2.1 presents models that can be used to select the checkpoint interval for an application and some of the fundamental assumptions made by these models, while Section 2.2 discusses the consequences of using improper checkpoint intervals. Section 2.3 presents the work of several groups on curve-fitting failure logs on different systems and Section 2.4 discusses related work on failure data clustering.

2.1 Analytical Models for Determining the Checkpoint Interval

As already discussed in Chapter 1, analytical models exist to determine the checkpoint interval for an application in terms of minimum execution time or minimum number of checkpoint I/O operations (Daly, 2006) (Arunagiri, Daly, & Teller, 2009). These models indicate that choosing an appropriate checkpoint interval is important to reduce checkpoint overhead associated with recomputation and checkpoint I/O operations. These models assume that failures exhibit an exponential distribution and that model parameters (such as restart time, checkpoint latency, etc.) do not change during application execution. As discussed in Section 2.3, previous work showed that failure data for several different systems do not explicitly follow an exponential distribution. However, there is conjecture that the failures that occur during a system's steady state do exhibit an exponential distribution. Of course, the applicability of these models depends on how well the system failure behavior fits an exponential distribution and how well the model's parameters match the behavior of the system and application.

Other methods utilize machine learning algorithms to determine when to checkpoint based on whether a failure is likely. Meta-learning, rule-based predictors using weighted polling have been successful in predicting some failure events (Gu, Zheng, & Lan, 2008). Other machine learning methods also have been employed with some success in predicting failures; these include Bayesian Networks, Decision Trees, and others (Zheng, Lan, Gupta, Coghlan, & Beckman, 2010) (Zheng, Lan, Park, & Geist, 2009) (Liang, Zhang, & Xiong, 2007). In comparing different methods of determining when to checkpoint Liu, Nassar, Liangsuksun, et al. [9] found that interval-based checkpointing typically outperforms such risk-based checkpoint methods. This made interval checkpointing preferable for the purposes of examination.

2.2 Impact of Checkpoint Interval

The impact of the checkpoint interval on application and overall system performance was investigated in the research conducted by Daly (Daly, 2006) and Arunagiri, et al. (Arunagiri, Daly, & Teller, 2009) They state the impact of inadequate checkpointing. They show that inadequate checkpointing can increase total execution time and even cause an increase in the number of defensive I/O operations when the checkpoint interval is too large.

Also, research by Jones, Daly, and DeBardeleban demonstrated the impact of improper checkpoint intervals. Due to difficulties in determining the parameters needed to compute the optimal checkpoint interval, it can be hard to identify the correct checkpoint interval to use. They showed the impact that faulty estimates of the checkpoint interval might have on overall performance: underestimates in the optimal checkpoint interval had a clear impact, reducing the overall efficiency of the program, but overestimates, as much as twice the actual optimal checkpoint interval, only resulted in an efficiency difference of less than ten percent (Jones,

Daly, & DeBardeleban, 2010). Moreover, tying this into the work done by Arunagiri, et al., if an application checkpoints more often than that frequency deemed optimal, this significantly increases I/O activity, potentially to the detriment of system performance; whereas overestimating the checkpoint interval will reduce the number of checkpoint I/O operations performed, potentially reducing the load on network file system. Thus, it is important to ensure that the parameters for computing checkpoint intervals are estimated or measured accurately.

In order to test some of the assumptions made by analytical models, it is necessary to collect performance data of applications that perform periodic checkpoint operations. Both Bidisha Chakraborty, another researcher in the HiPerSys Group at UTEP, and I concurrently focused on this objective; Chakraborty's work focused the RAxML-Light application. The process used by Chakraborty mirrors that followed by me: code was added to the application to collect I/O and execution time information and to simulate failures, and the application was run under different experimental scenarios on the Ranger HPC system at TACC. Chakraborty found that RAxML's checkpoint latency was very small and did not vary significantly with problem size. No strong conclusions could be drawn from her experiments about RAxML's execution time and checkpoint overhead due to large variations in execution times, which likely resulted from sharing of the I/O subsystem with other applications (Chakraborty, 2012).

2.3 Failure Distributions on LANL Systems

LANL (Los Alamos National Labs) released failure logs for 22 of their systems for the time period between 1996 and 2005 (Usenix - Computer Failure Data Repository (CFDR)). In order to better understand HPC system failures, analyses have been conducted on such failure records to determine patterns of failures. In these analyses, failures were evaluated for multiple

HPC systems collectively and for individual systems. In analyses conducted both by researchers at Carnegie Mellon University (Schroeder & Gibson, 2006) and separately by Bidisha Chakraborty [2], the collective failure distribution fits a Weibull distribution but does not fit an exponential distribution. Analyses of individual system-hardware failures by Chakraborty showed that only 9 out of 18 of the LANL systems had a good fit for a Weibull distribution and 4 out of 18 had a good fit for an exponential distribution (Chakraborty, 2012).

2.4 Clustering Failure Events

One of the key problems facing analytical modeling of failures is that most mathematical distribution patterns treat individual failure events as distinct entities unrelated to one another. There are clear circumstances under which failures are likely related. For example, if two nodes in the same rack fail within a short period of time of each other, both due to overheating, it is possible that the failure events were related based on environmental factors. Therefore, determining if failure events are either independent or related is an important part of failure research.

In efforts to automate the analysis of failures, some researchers have turned to system RAS (Reliability, Availability, and Serviceability) logs. These logs contain important information, e.g., warning, error, fatal, and failure messages, that give clues about the overall system state and the states of individual nodes. Efforts have been made to identify related failure events through use of associative rules, time windows (Gu, Zheng, & Lan, 2008), and methods like Neural-Gas clustering of failure messages (Hacker, Romero, & Carothers, 2009) (Zhong, Khoshgoftaar, & Seliya, 2004). These methods have been used on logs of BlueGene/L and

BlueGene/P (also known as Intrepid). The Neural-Gas method of clustering related failure events was uses to reduce clustered events to single failure events within these logs. This resulted in the failure distribution on these Blue Gene systems fitting a Weibull distribution, consistent with other, smaller systems. Although the Neural-Gas method was found to have a better mean squared score than K-Means algorithms in a comparison of several computer software datasets, the margin of false positives and false negatives was very close between the two different algorithms, with the better algorithm varying by dataset (Hacker, Romero, & Carothers, 2009). Moreover, the researchers noted that the mean squared difference may simply be the result of noise in the data (Hacker, Romero, & Carothers, 2009).

3. Methodology

This chapter presents detailed information regarding the programs and software tools used in this study for:

1. Measuring the checkpoint latency, execution time, and checkpoint/restart overhead of an application for different runs on a large-scale HPC system;
2. Simulating execution and failure of an application using the recorded measurements and using failure data collected from HPC systems; and
3. Running cluster analyses of failure data logs of HPC systems to locate potential failure clusters.

This includes discussion of available tools and identification of the tools that were ultimately used in this study.

3.1 Applications for Investigating Checkpoint Data: RAxML and NAMD

In order to investigate the relationship of wall-clock application execution time and number of checkpoint I/O operations as a function of the checkpoint interval and to test some of the assumptions made by the analytical models mentioned in Chapter 1, it is necessary to collect real-world data of applications that perform periodic checkpoint operations. Discussions with personnel at Texas Advance Computing Center (TACC) led us, the HIgh PERformance SYStems (Hipersys) group at UTEP, to two applications that were known to perform regular checkpoints and were known to be run frequently on Ranger: RAxML and NAMD (Browne &

Hammond, 2010). Having already briefly discussed Bidisha Chakraborty's work that collected checkpoint data for RAxML-Light (Chakraborty, 2012), the focus of this study is on NAMD.

3.1.1 What is NAMD?

NAMD (Not Another Molecular Dynamics) is a Molecular Dynamics simulator that performs its own checkpoint operations. NAMD has an iterative execution, where it runs a given simulation a number of steps, which are called "timesteps". The value of the number of timesteps is set in the simulation's configuration file. NAMD can be configured to store the state of the simulation at multiples of a fixed number of iterations so that it can resume execution from these points. When a checkpoint step is reached, the main process receives the checkpoint data from the other processes. Once all the data is received from the other processes, the main process writes the data to three files which store different data (configuration, positions, and velocities). This is an asynchronous checkpoint process, where individual processes send the data to the main process when they reach that checkpoint step, then resume computation without waiting for the other processes to reach the same point (Bhandarkar, et al., 2010).

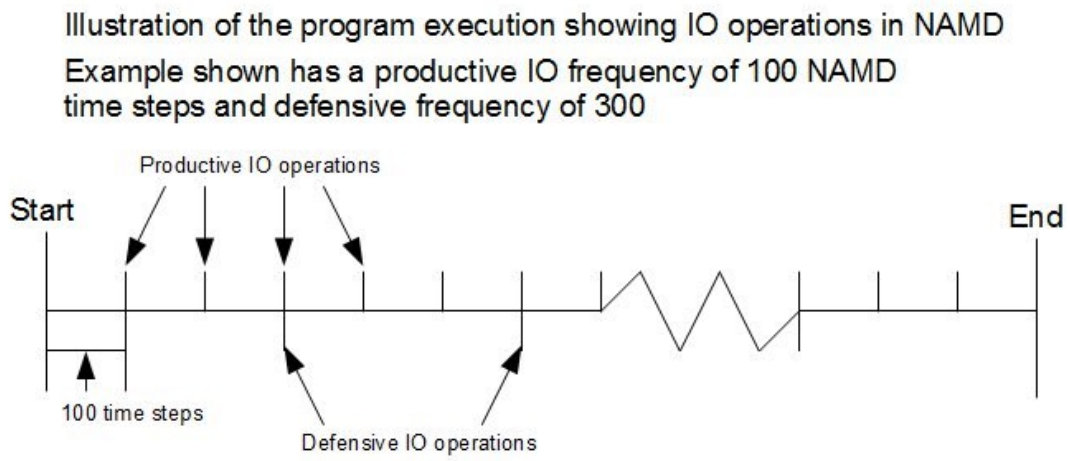

Figure 3.1 Illustration of Defensive and Productive I/O Operations in NAMD.

3.1.2 The Importance of NAMD

NAMD is a community code that is widely used in visualization of biochemical processes (Philips, et al., 2005). As a community code, it is supported by a wide range of people in varying specialties, including biochemistry, biophysics, etc. As previously explained, it has checkpointing capabilities that are often utilized by its users. It is one of the most frequently run codes at TACC (Browne & Hammond, 2010), and as such, is a strong point of interest for collection of checkpoint latency, execution time, and other information.

3.1.3 Existing Methods for Collection of I/O Information in NAMD

Darshan is an I/O monitoring software that collects information about the I/O operations that an MPI program performs. It can track the size of, times of, and time length of each file write. Although Darshan is designed to monitor MPI programs, it can collect some data on POSIX writes as well, as long as the program is run in an MPI environment. Darshan, however, has limitations. It cannot distinguish between productive and defensive I/O, and cannot determine the time invested in preparing checkpoint files, i.e., it does not tell you how much time the program spent organizing data before writing the file. For example, in checkpointing applications such as DMTCP, a large portion of the checkpoint latency is spent synchronizing the processes and collecting the contents of the system memory of the processes for the checkpoint (Ansel, Arya, & Cooperman, 2009). For DMTCP, relatively little time is spent actually writing the file, so Darshan cannot provide a complete picture of checkpoint latency. It was necessary to collect as complete a picture as possible of the time spent by NAMD in processing a checkpoint in order to get an accurate picture of checkpoint latency, not just the time spent writing the files. Due to these reasons, we had to modify the NAMD code to collect checkpoint latency data.

3.1.4 Modifications Made in NAMD's Code to Collect Latency Data

In order to collect data on checkpoint latency in NAMD, it was necessary to add instrumentation to the code. NAMD writes three files per checkpoint: one file stores the conditions and parameters; one stores the positions of the atoms in the simulation; and the last stores the velocities of the atoms in the simulation. If this was a sequential process, this would

simply be a process of recording the times before the checkpoint process starts on the first file and after the process finishes processing the third file. However, early testing revealed that the writing of the files is not sequential. Usually the parameters file is written first, but sometimes it is not. Because of this, the code was instrumented to mark the beginning and ending of each of the functions that generate the three files produced for each checkpoint. The latency was then determined by the start of the function of whatever file wrote first and the ending of the one that finished last. The full details of the changed code can be found in Appendix A.

3.1.5 TCL Scripting for Simulating Failures in NAMD

In order to investigate how the execution time and defensive I/O of NAMD would be affected by failures of a known distribution, it was necessary to add the capability to simulate failures in NAMD. NAMD has the capability to run a limited TCL script in the input file that sets the parameters for the simulation. This capability was exploited to implement the simulation of failures in NAMD. The choice to implement this as part of the simulation's input file was made so that the failure simulating code could be run on unmodified NAMD should it be needed or desired. The script simulates failures and restarts by pulling random values of a known distribution from a file. These values determine when each node is going to simulate failure. The simulator estimates the number of NAMD timesteps needed before the next failure is reached and executes for that number of steps. Thereafter the content of the checkpoint files are read and the application run is restored to the state it was in at the last stored checkpoint, and the process repeats until the application terminates. Figure 3.2 shows a flowchart of the process. The full code can be found in Appendix B.

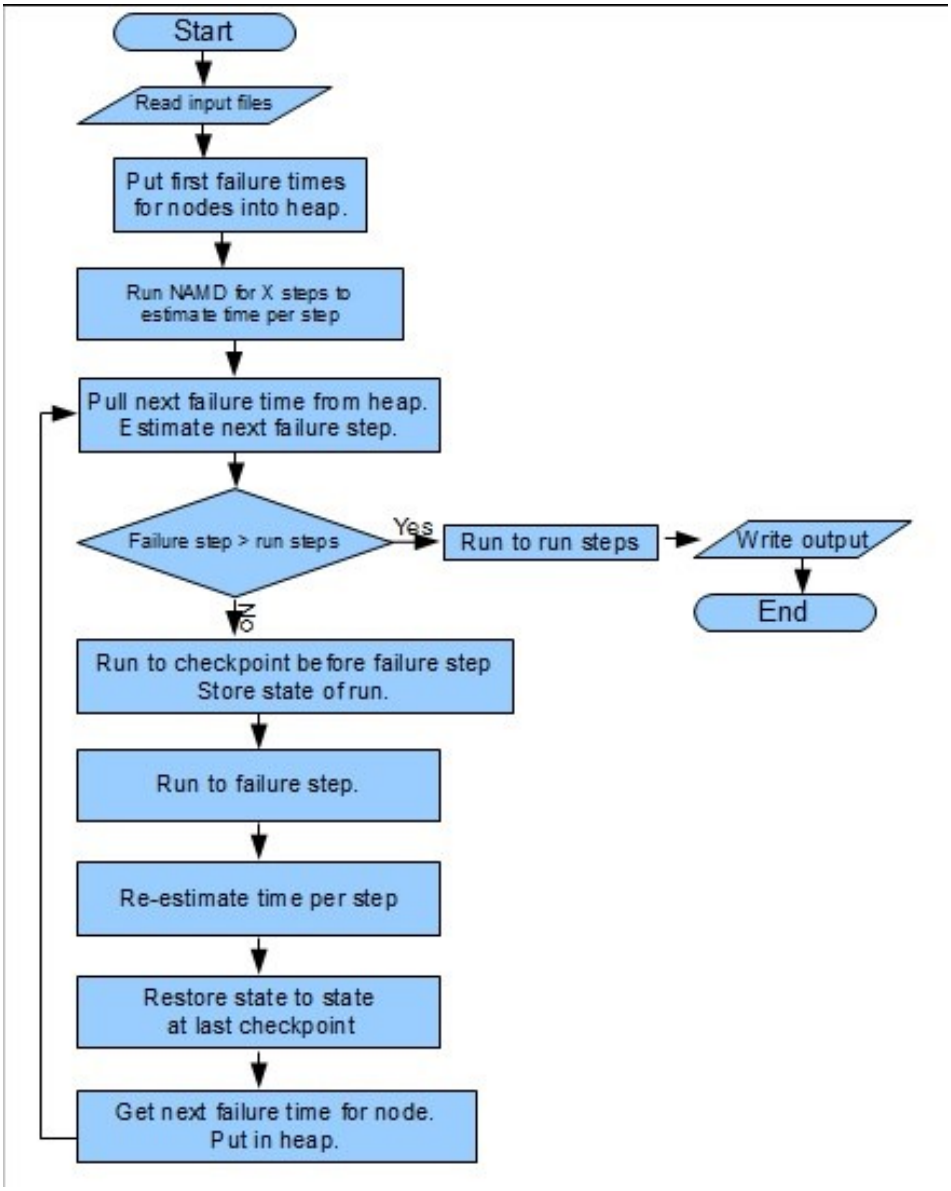

Figure 3. 2 NAMD Checkpoint Simulator Flowchart.

3.1.6 Experimental Platform

Experiments were conducted on Texas Advanced Computing Center's Ranger system. Ranger consists of 3,936 nodes, each with four AMD Opteron quad-core processors, and has 32 GBs of memory. Different NAMD tests were conducted across single and multiple nodes on this

system to collect checkpoint and execution time data. More details on specific experiments are presented in Chapter 4.

3.2 Simulation of Failures Based on Real-World Failure Data

Researchers have found that, in some systems, failures do not follow exponential distributions. Because of this, it was important to run simulations with real failure distributions instead of failure distributions that followed a statistical function. The choice to simulate the execution was made because running real-world tests on a system like Ranger would be prohibitively expensive in terms of the computational resources needed to run the variety and scale of tests desired.

3.2.1 Failure Data Used

The ideal data would identify individual failures and categorize them according to the cause of the failure. Multiple failure logs are available from the Usenix Computer Failure Data Repository. The types of failure logs vary from hard disk replacement logs, hardware service logs, to HPC failure and RAS logs (See table 1). With RAID filesystems, hard-disk failures do not necessarily mean that a running application would fail as well. This rendered hard-disk related logs useless. Service logs were mostly for ISPs, not HPC systems, and recorded service times, not failure times. This made service logs useless in this inquiry. Of the candidates, the failure logs from LANL and PNNL, and the RAS logs from Bluegene/L and Bluegene/P were of most interest (see table 2).

Table 3. 1 Failure Log Data from CFDR Evaluated ror Use for Simulation and Clustering.

Name	Description	Considered? Why/Why Not
LANL	Failure data from 22 systems.	**Yes**. Records failures and causes.
HPC1	Hardware replacement logs.	No. Too narrow in scope.
HPC2	Disk replacement log.	No. Too narrow in scope.
HPC3	Disk replacement log.	No. Too narrow in scope.
HPC4	RAS event logs from 5 systems.	**Yes**. BlueGene/L log considered.
PNNL	Failure logs for a single system.	**Yes**. Records failures and causes.
NERSC	I/O failure data.	No. Too Narrow in scope.
COM1	ISP hardware replacement logs.	No. Not an HPC system, too narrow.
COM2	ISP Warranty hardware replace.	No. Not HPC, too narrow.
COM3	ISP hard drive replacement logs.	No. Not HPC, too narrow.
Cray Data	Event + system logs for cluster.	No. Failures not identified in logs.
Intrepid	RAS log for BlueGene/P.	**Yes**. Failures identified in log.

Table 3. 2 Properties of Logs Considered For Use in Simulation and Clustering

Data Set	Nodes	Systems	Messages	Failures
LANL	5261	22	23,614	23,614
PNNL	980	1	2,944	2,944
Intrepid	40,960	1	2,084,393	33,370
BlueGene/L	66,596	1	4,747,963	348,460

The LANL data was chosen for testing purposes. The reasons for this are numerous, including: the logs were separated into 6 failure categories: Facilities, Hardware, Human Error, Network, Software, and Unclassified; each failure was identified and documented by the people actually maintaining the systems, meaning each failure message in the log was a unique failure event, unlike RAS logs, where multiple messages were identified for each failure; and, lastly,

the LANL logs had data for 23 systems, allowing tests to be conducted using data from different systems so that the testing is not dependent on a single distribution or upon artifacts that may be present in a single system. These factors made the LANL failure logs preferable to the other logs.

3.2.2 The Importance of Categorized Failures

As was discovered in work by Chakraborty, hardware and software failures do not necessarily follow the same distributions (Chakraborty, 2012). Because of this, testing based on separate categories of failures would be more likely to reveal patterns. Therefore, the tests in this study focused on hardware failures, which were the most common category of failures in the LANL data. Maintenance activity, which was also listed as failures in the logs, is ultimately controlled by the personnel at the facility, not by hardware reliability, so it was stripped from the logs before the data was used in our study.

3.2.3 LANL Data Used for the Simulation

The data used for the simulation was from LANL systems 18 and 19. These were chosen on the basis that they had a large number of nodes, they both had more than three years of data, and both failed tests conducted by Bidisha Chakraborty, showing that both exponential and Weibull distributions were not good fits for the failure distributions (Chakraborty, 2012). The fact that standard distributions did not fit well for these systems was considered a positive feature for the tests that were to be conducted because failures that fit standard distributions are easily

modeled analytically. It was important to see what would happen in terms of execution time, optimal checkpoint, and defensive I/O operations for systems that did not explicitly match the expected distributions.

3.2.4 How the Simulator Works

The simulator was programmed to take a list of jobs and a failure file, and to simulate execution of checkpoints, failures, and restarts based on a number of parameters. The simulator code can be found in Appendix C. The simulator is event driven. It uses a priority queue, but allows for backfill. Backfill is a process whereby the simulator estimates when adequate resources will be available to run the highest priority job and runs lower priority jobs on the already available resources if those jobs will complete before the necessary resources to run the highest priority job are estimated to free. When simulating the run of a job, the simulator assigns jobs to the number of nodes specified for that job. It then adds checkpoint, failure, and completion events for the job to the event heap based on what will happen next. The job continues to execute on the nodes, periodically experiencing checkpoint events until completion or failure occurs. When a failure occurs, the job is removed from the nodes on which it is running, the resources are freed, and the job is placed back on the job heap along with its computational progress at the point at which the last checkpoint occurred. Since the simulator pulls failure times from a file, it is possible to test what will happen with failure data of any distribution.

3.2.5 Job Data Used for the Simulator

The parameters used for the jobs that were input to the simulator were parameters derived from the tests run on Ranger. The parameters needed for the jobs are number of nodes, start/restart time, checkpoint latency, and solution time. The number of nodes chosen for the simulation studies was 256. This choice was based on discussions with TACC personnel and NAMD scalability studies. This number was chosen to represent large NAMD jobs run on Ranger. Start time is the time it takes from the start of execution until NAMD begins the first timestep. This value is reported by NAMD in the output at the start of a NAMD simulation. The value used, 30 seconds, was based on the highest value encountered in the totality of the tests conducted. The checkpoint latency was collected through running tests on Ranger using the modified NAMD code, which was described in section 3.1.4 The average value was used. The solution time used was 24 hours; this value was chosen for a couple of reasons. Although the tests that were run at TACC were targeted for approximately 20 hours, the tests conducted strongly suggest that the execution time scales linearly with respect to the number of timesteps NAMD is to execute. With this knowledge, even though the longest 256 node test that was run (results can be found in the Appendix) took less than two hours, it is possible to model a job with the same simulation parameters, but more timesteps. This allowed complete flexibility in terms of the solution time chosen. Because of this flexibility, 24 hours was chosen because TACC's job running system implements a 24-hour limit for processes running on the normal queue. By making the solution time 24 hours this tests limits imposed in an actual HPC data center.

3.3 Clustering

On the topic of failure distributions, it is often difficult to determine if failure events are independent of each other or related to each other. If some failures are related to each other, it may be possible to predict some failure events in advance, allowing programs running on nodes that are predicted to fail to perform additional preventative checkpoints to reduce computation loss. Clustering is a process to try to determine the relatedness of data points, such as the failure events. Clustering was performed using WEKA machine learning program on real-world failure data to see if patterns of related failures might be determined. As Zheng, et al. have shown, clustering failures can often result in clearer failure distributions (Zheng, Lan, Park, & Geist, 2009).

3.3.1 EM and K-Means with Density Cross Validation Clustering

There are many different clustering methods available in WEKA. Each has different strengths and weaknesses. Expectation-Maximization (EM) is a clustering algorithm that tries to fit the data points to Gaussian distributions (Gupta & Chen, 2010). This method is good for detecting when data points are clustered based on statistical distributions. This method was used to find clusters that might have very clear patterns.

K-Means is an often-used method of clustering. This method starts by either algorithmically or randomly assigning initial values for the means. Points are assigned to the cluster associated with the mean closest to that point based on Euclidean distance. After points are reassigned, the mean is recomputed based on the points assigned to each group. The process

repeats until no points are reassigned or a maximum number of iterations are reached (K-Means Clustering - Wikipedia). Simple K-Means is not a density-based clustering algorithm though. Density-based clustering allows points to be reassigned based on their proximity to a cluster with a mean that is further away than another, but the member points of which are closer than the member points of the other cluster. Thus, density cross validation was used with the K-Means clustering to achieve better clustering. These clustering methods were used with the LANL failure data to try to find patterns of spatio-temporal clustering.

4. Experiments and Results

With the goal of understanding the execution time of and number of defensive I/O operations generated by real applications as a function of checkpoint interval, we performed experiments on the Ranger HPC system at TACC and event-driven simulations, both of which were driven by NAMD. In addition, this study includes an analysis of LANL failures logs from CFDR to investigate the presence of clusters, which could potentially point to predictable failure patterns.

As explained in Chapter 1, code modifications were required to collect execution time and checkpoint performance data on an HPC system and to inject failures. The modifications and the subsequent experiments, conducted both locally at UTEP on a system with dual-core processors and remotely on nodes of Ranger at TACC, are discussed in this chapter, along with the experimental results.

As changes were made to the NAMD code, the code was tested locally at UTEP on a dual-core processor system. Section 4.1 describes the testing and the result of the testing, which verified the correctness of the code and the impact of the modifications on NAMD performance. The initial testing of the modifications to the logging function of NAMD was performed on Ranger; the testing and results are presented in Sections 4.2 and 4.3. Section 4.2 presents the testing of the modified NAMD code driven by the APOA1 dataset, which verified that the modified NAMD code works on large-scale systems and has minimal overhead. And, Section 4.3 presents the tests driven by NAMD with the STMV dataset, which were used to collect preliminary data on execution time per NAMD timestep, checkpoint latency, program startup time, and other information required for running larger experiments.

As described in Chapter 1, TCL scripts were used to simulate failures (inject simulated failures) in NAMD runs. Sections 4.4 and 4.5 present tests of these features driven by the SMTV dataset and executed on Ranger. Section 4.4 presents experiments that check the correctness of the code, i.e., verify that it performs the designed task, and report the performance overhead introduced by the modification. Section 4.5 presents the set of final experiments conducted with the modified NAMD code that uses failure times generated by a statistical distribution function. These final experiments were conducted with different checkpoint intervals and for each experiment the following data was collected: execution time, average checkpoint latency, startup time, and number of checkpoint I/O operations. To study the behavior of execution time and number of I/O operations when periodic checkpointing applications are executed at scale, Section 4.6 presents the results of event-driven simulations when NAMD is executed on 256 nodes. Finally, Sections 4.7 and 4.8 present a cluster analysis of LANL failure data using E-M clustering and K-Means with density cross-validation clustering, respectively.

4.1 Verification of Correctness and Performance Impact of NAMD Code Modifications

It was essential to run experiments to make sure that (1) the changes to the logging functions of the NAMD code correctly measured and collected the desired data and (2) the code did not incur undue performance overhead. To accomplish this, the program was run on a stand-alone system comprised of an Intel Xeon dual-core processor with simultaneous multi-threading, which supported the concurrent execution of four NAMD threads. The APOA1 dataset was used for these experiments, which doubled as preliminary tests of the larger-scale experiments conducted on Ranger.

4.1.1 Experiments

Apart from the fault injection code, two functions were added to NAMD: (1) the first enhances the logging function to collect data such as execution time, checkpoint latency, and number of checkpoint I/O operations and (2) the second performs the checkpoint operations, which write the checkpoint or restart file. The purpose of the experiments described below is to verify that the functions added to enhance the logging function and write the restart file worked as expected, were not causing undue performance overhead, and collected preliminary restart latency data for NAMD. Three sets of experiments, described below, where run. Each experiment in every set was composed of five trial runs, each executing an instance of NAMD. The first set of experiments, comprised of three subsets, were intended to determine the performance impact of the code modifications on NAMD execution time. To obtain the baseline for comparison, the first subset of experiments was run without checkpointing and without the logging functions. The second subset was run with checkpointing and without the logging functions. And, the final subset was run with checkpointing and the logging functions. For runs with checkpointing, the restart files were written every 20 NAMD timesteps. Execution times were collected for the execution of five complete uninterrupted NAMD runs. These experiments were run on 1, 2, 3, and 4 concurrently executing threads and execution times were collected for each run.

The second set of experiments was conducted to test the ability of the code to collect the checkpoint latency data. Checkpoint files were generated every 20 of the 500 NAMD timesteps and the average latency was computed for each run. These experiments were run on 1, 2, and 4 concurrently executing threads.

The third set of experiments was conducted to determine the performance impact of the checkpoint write operations on total execution time. Four-thread runs of NAMD were executed for 500 timesteps with five different checkpoint intervals measured in terms of the number of timesteps between checkpoint writes: 20, 40, 60, 80, and 100. Five trial runs were executed for each checkpoint interval.

4.1.2 Results

Note that every experiment reported below has five trial runs and the numbers reported are averaged over the five runs. The results of the first set of experiments show that the logging functions either did not significantly impact the execution time of NAMD or their execution time was not discernible given the data from the five runs. As depicted in Table 4.1 below, the average execution time with checkpoint writes and logging was sometimes less (by 3 seconds), sometimes equal, and sometimes greater (by 2 or 5 seconds, i.e., less than .2% or .6%).

Table 4. 1 Total Average Execution Time of NAMD Runs

Threads	No checkpoint writes, no logging	Checkpoint writes, no logging	Checkpoint writes, logging
1	1,527 s	1,528 s	1,524 s
2	790 s	793 s	792 s
3	880 s	873 s	875 s
4	674 s	673 s	674 s

The results of the second set of experiments, which are presented in Table 4.2, show that the modified NAMD was able to collect the start and end times of checkpoint writes. This data also indicates the checkpoint latency of a single-node NAMD execution. The checkpoint latency decreases from 28 ms to 24 ms when the number of threads per NAMD run changes from one to

two. It is unclear why the latency was less with two threads, the 4 ms difference may just be the result of normal variance in execution time. However, when the number of threads increases to four, the checkpoint latency seems to increase substantially from an average of around 24 ms to an average of 109 ms. Also, column two of the table shows that checkpoint writes are performed more frequently as the number of threads per NAMD run increases, going from 61 seconds between checkpoints to 28 seconds between checkpoints. Since the number of NAMD timesteps is fixed at 500, this means that there is a reasonable amount of parallelism that can be exploited by the four threads. These results suggest that on this particular dual-core SMT system, four-thread runs may not show an accurate representation of checkpoint latency due to the non-proportional growth in checkpoint latency.

Table 4. 2 Relationship between the Number of Threads and Checkpoint Interval and Latency for NAMD with Timestep=500.

Threads	Measured Checkpoint Interval in terms of Wall-clock Time in secs	Checkpoint Latency in ms
1	61.1	28
2	32.4	24
4	28.2	109

The results of the third set of experiments, shown in Table 4.3, is that the overall impact that checkpointing had on these single node execution times of NAMD was negligible as average execution time varied only by two seconds between the run with five checkpoint writes and the run with 25 checkpoint writes, a variance that accounts for only 0.03% of the execution time.

Figure 4. 3 Total Average Execution Time of NAMD Runs with a Different Number of Checkpoint File Writes.

Checkpoint Interval	20	40	60	80	100

in terms of Timesteps					
Number of Checkpoint Writes	25	12	8	6	5
Total Execution Time in secs	673	673	673	672	671

4.2 NAMD Driven by opoa1 Dataset: Preliminary Experiments on Ranger

Preliminary testing was conducted on Ranger using the APOA1 dataset to verify that the code worked as desired in multi-node runs. As mentioned previously, three files are generated when NAMD checkpoints. Initial testing of the modified NAMD code revealed an occasional discrepancy in the checkpoint data collection, where the third of those checkpoint files was sometimes written before the first. This indicated that something that was thought to run in series was actually running in parallel. Because of this, the instrumentation was changed so that more data was collected by the logging functions, i.e., the functions were changed to record the time before and after each of the files of a checkpoint was written. The sum of the data for all three file writes is the total latency. Once that problem was resolved, tests were run to check the consistency of execution times, the scalability of NAMD, the scalability of the checkpoint latency, and the overhead of the modified code.

4.2.1 Tests Conducted

Tests were run on 1, 2, and 4 nodes with the APOA1 dataset for 30,000 time steps. The checkpoint interval was set to a value of 5,000 so that six checkpoints would be generated.

4.2.2 Results

Table 4.4 and the graph in Figure 4.1 show the average execution times of the runs with two repetitions of each run. As can be seen, there is no significant overhead introduced by the checkpoint logging functions, with the largest difference between checkpoint without logging and checkpoint with logging was less than 2%. Additionally, the difference between runs with the same parameters showed sometimes showed greater variance, 2.7%. Another observation is that as the number of cores or nodes that NAMD was run on was doubled, the wall-clock execution time is not quite halved; thus, it is clear that the more nodes that the process is run on, the more total processing time is taken, which is probably due to network scaling costs.

Table 4. 4 Average Execution Time of NAMD with APOA1 Dataset Executed on Ranger in Seconds

Number of Processors	No checkpoint writes, no logging	Checkpoint writes, no logging	No checkpoint writes, logging	Checkpoint writes, logging
16	4,660.5	4,639.7	4,628.3	4,641.0
16	4,647.3	4,648.3	4,639.1	4,648.2
32	2,489.4	2,497.7	2,484.6	2,492.9
32	2,515.8	2,501.3	2,516.6	2,476.1
64	1,472.5	1,391.3	1,414.5	1,417.5
64	1,415.5	1,431.1	1,419.9	1,438.3

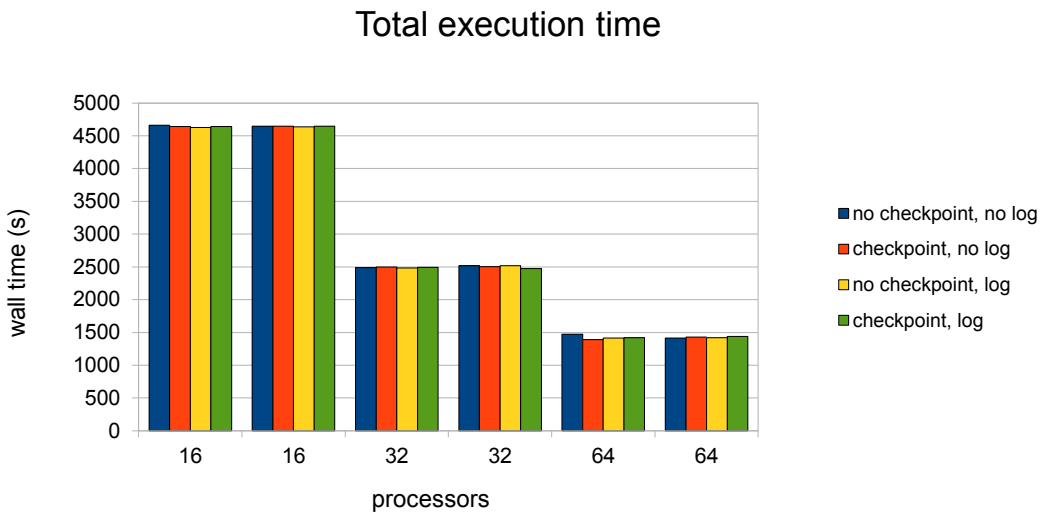

Figure 4. 1 Average Execution Time of NAMD with APOA1 Dataset Executed on Ranger

Another observation from the plot in Figure 4.2 is that execution time grows linearly with the number of NAMD timesteps. The graph shows the wall clock time in milliseconds when a checkpoint operation occurred. This observation of linear scaling with respect to NAMD timesteps forms a foundation for other assumptions made in later experiments.

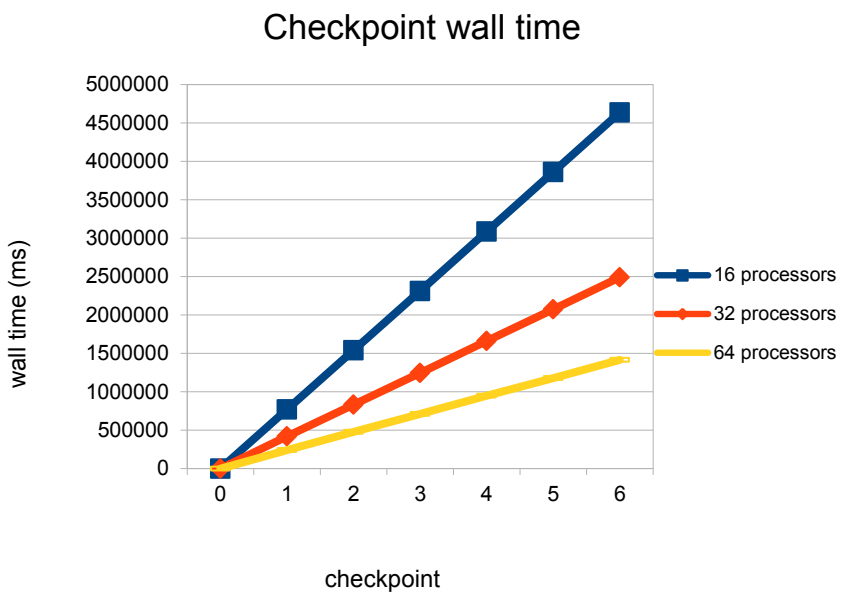

Figure 4. 2 Wall Clock Time (in milliseconds) when Checkpoint Operations were Performed by NAMD with APOA1 Dataset Runs on Ranger

Another important piece of information collected was the latency of checkpoints. For NAMD driven by the APOA1 dataset, the value of the latency varied between 0.015 seconds and 0.25 seconds. Given the desire to run 20-hour experiments with a simulated MTTI of 6, 8, and 10 hours, this was problematic because the optimal checkpoint interval computed using Daly's estimate (Daly, 2006) is about 20 seconds, which implies that we need to checkpoint as frequently as three times per minute. TACC personnel had previously expressed concerns about frequent file writes, including NAMD checkpointing, causing problems with the Lustre file system. Therefore, we looked for a dataset with a larger footprint and possibly a larger checkpoint latency so that the checkpointing frequency could be reduced. Because of this, testing was moved to the STMV dataset.

4.3 NAMD Driven by STMV Dataset: Preliminary Experiments on Ranger

Preliminary experiments were run to collect checkpoint latency information to guide the experimentation described in section 4.4 and to estimate the average wall-clock time per NAMD timestep. These experiments consisted of running the STMV dataset for 10,000 timesteps on four nodes (64 cores) of Ranger at TACC. The average wall-clock time per NAMD timestep was needed to determine the number of timesteps that would comprise a job that would take approximately 20 hours to run on four nodes of Ranger.

4.3.1 Experiments

NAMD driven by the STMV dataset was run on for nodes of Ranger for 10,000 NAMD timesteps with the checkpoint interval, in terms of NAMD timesteps, equal to 100. Such a run produced a total of 100 checkpoint file writes.

4.3.2 Results

The data collected across 100 checkpoints showed that checkpoint latency varied between 0.12 seconds and 0.28 seconds, with the average value being 0.164 seconds. Because of the longer latency time, checkpointing would be conducted less frequently. This was a better latency value for the purposes of the experiments that were to be run because writing checkpoint files too often would place a burden on the network I/O system. It also was found that the STMV dataset executed at a rate of about 2.154 timesteps per second.

4.4 Testing of TCL Script for NAMD Failure Simulation on Ranger

Tests were conducted with the STMV dataset to verify the correct execution of the TCL failure simulating script. Execution log files indicate that the script executed as expected with the failure distribution files used.

4.5 Final Experiments: Large Runs of NAMD using the STMV Dataset and TCL Failure Simulating Script

The goal of this set of experiments was to study the behavior of execution time and number of defensive I/O operations of NAMD as a function of the checkpoint interval. Preliminary data collected from the experiments described in Section 4.3 were used to configure jobs with an estimated execution time of approximately 20 hours and with the desired checkpoint interval that was to be based on Daly's estimated model of the optimal checkpoint interval based on the MTTI, restart, and latency values observed.

4.5.1 Checkpoint Data for NAMD Driven by the STMV Dataset on Ranger

As shown in Table 4.5, a total of 72 NAMD experiments were conducted to collect execution time, number of checkpoint I/O operations generated, checkpoint latency, and the recomputation time in terms of the additional number of NAMD timesteps executed due to failures. These experiments were subdivided into three subsets of 24 experiments each; each subset assumed one of three different values of MTTI, i.e., 6, 8, and 10 hours. Each subset was used to study the number of defensive (checkpoint) I/O operations generated and the execution time for NAMD run with four different values of checkpoint intervals: half the optimal

checkpoint interval, the optimal checkpoint interval, two times the optimal checkpoint interval, and four times the optimal checkpoint interval.

Using data from the preliminary experimentation with the STMV dataset (described in Chapter 3), Daly's model was employed to estimate the optimal checkpoint interval. Failure times used as input to the TCL failure simulating script were computed using an exponential failure distribution. Since preliminary experimentation revealed the checkpoint latency to be from 0.12 seconds to 0.26 seconds, the average, 0.168, was used in computing Daly's optimal checkpoint interval. (Note that an interval analysis revealed that the entire range of optimal checkpoint intervals still fell within the range of the computed ½ optimal to 4x optimal checkpoint interval).

Table 4. 2 Details of the 72 Final NAMD Experiments on Ranger with the SMTV Dataset.

72 Experiments	MTTI	Checkpoint Latency	Repetitions
Subset 1	6 hrs	½ optimal*	6
	6 hrs	optimal	6
	6 hrs	2 x optimal	6
	6 hrs	4 x optimal	6
Subset 2	8 hrs	Same as for Subset 1	Same as for Subset 1
Subset 3	10 hrs	Same as for Subset 1	Same as for Subset 1

Even though the MTTI values used in these experiments are unrealistically small for four nodes, Daly's model is scalable in terms of time units (seconds, minutes, hours). This means that if all time values are multiplied by the same factor, the equation still holds.

4.5.2 Results

As can be seen in Figure 4.3, for the six repeated runs, the number of NAMD timesteps that needed to be recomputed due to failures increased as the checkpoint interval increased. This was consistent with expectations.

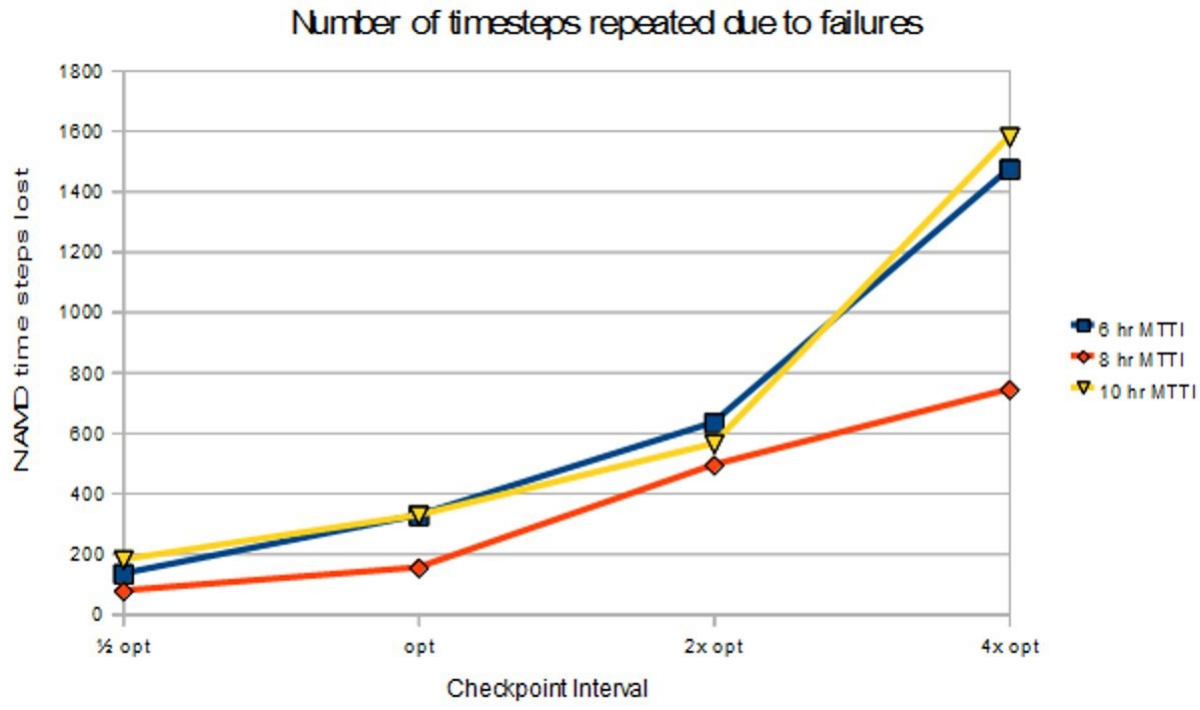

Figure 4. 3 Average Number of NAMD Timesteps Repeated Due to Simulated Failures.

Table 4. 7 Timesteps that Needed to be Repeated Due to Simulated Failures

MTTI	½ opt	Opt	2x opt	4x opt
6 hrs	133	326	633	1,473
8 hrs	76	153	493	743
10 hrs	180	326	563	1,580

Figure 4.4 shows the number of defensive I/O operations generated, averaged over the six repeated runs executed with the indicated parameters, as a function of the checkpoint interval. We see a clear reduction of I/O operations as the checkpoint interval increases, which is consistent with expectations and the analytical model.

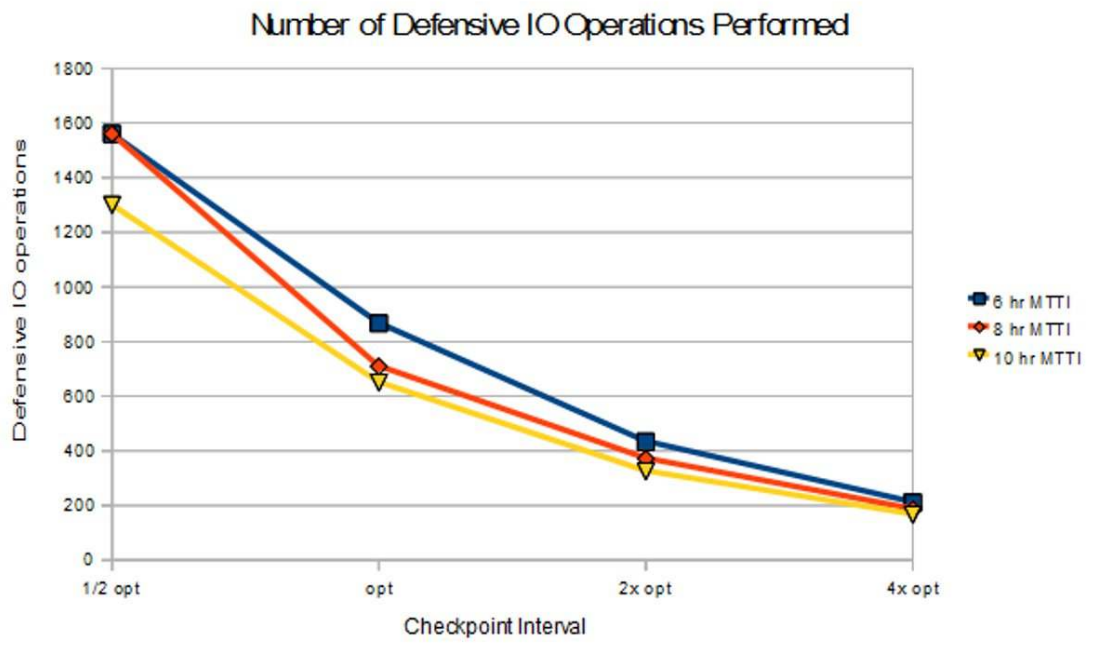

Figure 4. 4 Average Number of I/O Operations Generated in NAMD with Simulated Failures

Table 4. 8 Average Number of I/O Operations Generated in NAMD with Simulated Failures

MTTI	1/2 opt	opt	2x opt	4x opt
6 hrs	1,563	870	437	214
8 hrs	1,561	710	373	187
10 hrs	1,303	653	328	169

Figure 4.5 shows the execution times for NAMD experiments with simulated failures and restarts, averaged over the six runs. The trend does not explicitly match the expectation of the analytical model in terms of being a convex function with a single minimum. In fact, only in the experiments that assume a six-hour MTTI does the optimal checkpoint interval result in the best execution time. Note that baseline NAMD experiments were conducted without checkpointing and without simulated failures, and the variance in the execution times ranges from 19.88 hours to 20.44 hours. When this variance of about 3% was discussed with TACC personnel, they were not surprised. Because of the variance present in the runs, it is difficult to draw any conclusions from this data other than that the variance likely had a larger impact on the execution time than the overhead introduced by checkpoint and failures. It was expected that the variance would average out over many trials, however, it was infeasible for us to run the needed number of trials.

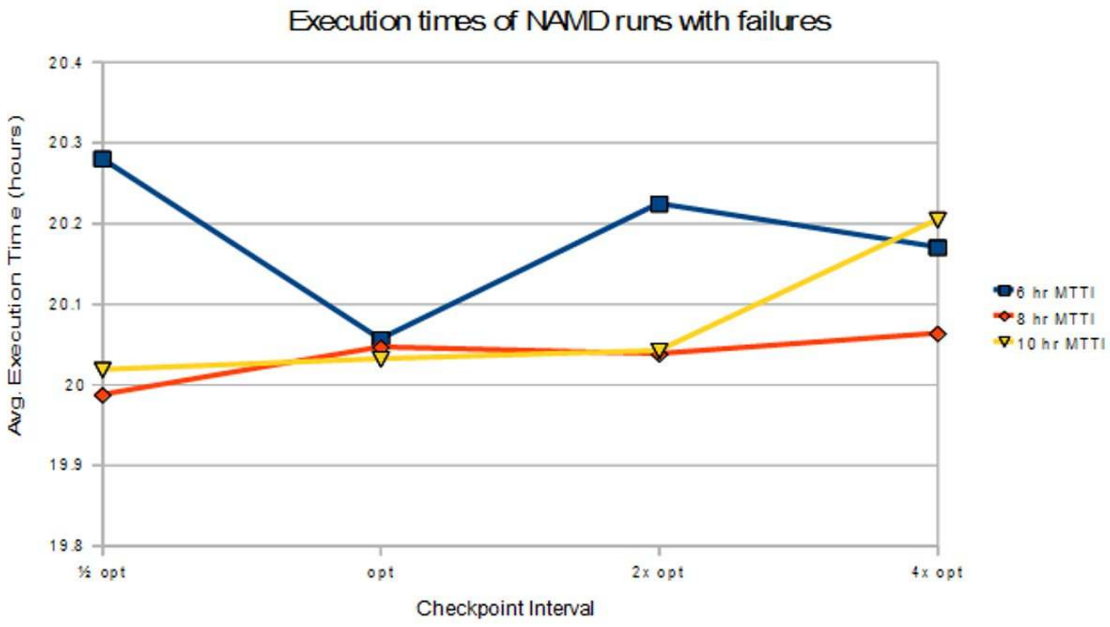

Figure 4. 5 Average Execution Time of NAMD with Simulated Failures

Table 4. 8 Average Execution Time of NAMD with Simulated Failures

MTTI	½ opt	Opt	2x opt	4x opt
6 hrs	73,007 s	72,204 s	72,807 s	72,608 s
8 hrs	71,952 s	72,166 s	72,135 s	72,227 s
10 hrs	72,065 s	72,113 s	72,149 s	72,735 s

4.6 Simulation of NAMD using LANL Failure Data.

As discussed previously, TACC personnel indicated that it is not uncommon for there to be a variance in execution time of 5% (Barth, Brown, Hammond, & Phillips, 2012). Because of this, a very large number of runs would be needed to average the variance out of the results. Due to the massive amount of time it would take and the resources that it would consume, it would be prohibitive to run large number of tests on Ranger to collect the needed data to determine definitively what performance impact might be incurred from an inappropriate choice of checkpoint interval. Because of this, it became essential to run simulations. In order to better reflect a real-world scenario, these simulations used CFDR failure data collected by LANL (Usenix - Computer Failure Data Repository (CFDR)) to determine when nodes failed and when they went back online. The data used was from systems where failure data did not show a good fit for either exponential or Weibull distributions (Chakraborty, 2012). Also, the value assigned to the checkpoint latency parameter was obtained from NAMD runs on Ranger.

4.6.1 Simulations

These simulations were conducted to quantify the impact that real-world failure data would have on the checkpoint/restart performance. Failure data from LANL and application data collected about NAMD from runs on Ranger were used for the simulations. The failure data

used was from LANL systems 18 and 19. Tests were repeated for both systems. For each system, 2,400 identical 24-hour jobs, which were based on the NAMD measurements previously collected, were simulated. For each job separate simulations were conducted for checkpoint intervals (all times in minutes): 1, 2, 3, 4, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 70, 80, 90, 100, 110, 120, 140, 160, 180, 200, 220, 240, 300, 400, 500, and 740. One simulation was run with no checkpointing.

4.6.2 Results

The graph for the NAMD simulations driven by LANL failure data for system 18, figure 4.7, shows the average of the execution times for all 2,400 simulations with a given checkpoint interval. This value is graphed along with the expected execution time according to Daly's analytical model to indicate how the execution time of a simulation with failures from a real non-exponential failure distribution compares with that predicted by the analytical model, which assumes an exponential failure distribution. As can be seen, Daly's analytical model for execution time tends to underestimate the values slightly, but within a very close margin (within 1 minute) of the simulated execution time for most values of the checkpoint interval. These graphs clearly show the optimal checkpoint interval to be a value between 3 and 4 minutes. Daly's analytical model for the optimal checkpoint interval places the optimal at 3.45 minutes for both systems with the checkpoint latency value used. Tests with system 19 showed the same trends.

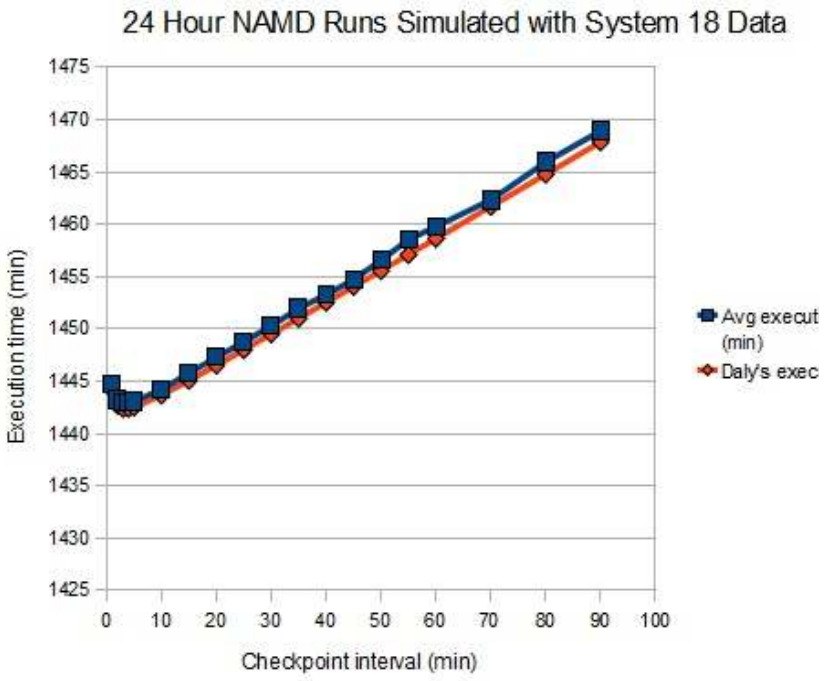

Figure 4. 6 Execution Time vs. Checkpoint Interval for NAMD Simulations for System with 2,400 nodes and LANL Failure Data from System 18

Note also that Figure 4.7, the graph of the simulated execution time for the 2,400 NAMD simulated runs for system 18, shows that even though the optimal checkpoint interval is between 3 and 4 minutes, it can easily be increased to a larger time, e.g., 8 minutes, with a minimal increase in overall system overhead. This illustrates the impact of the choice of checkpoint interval on the whole system.

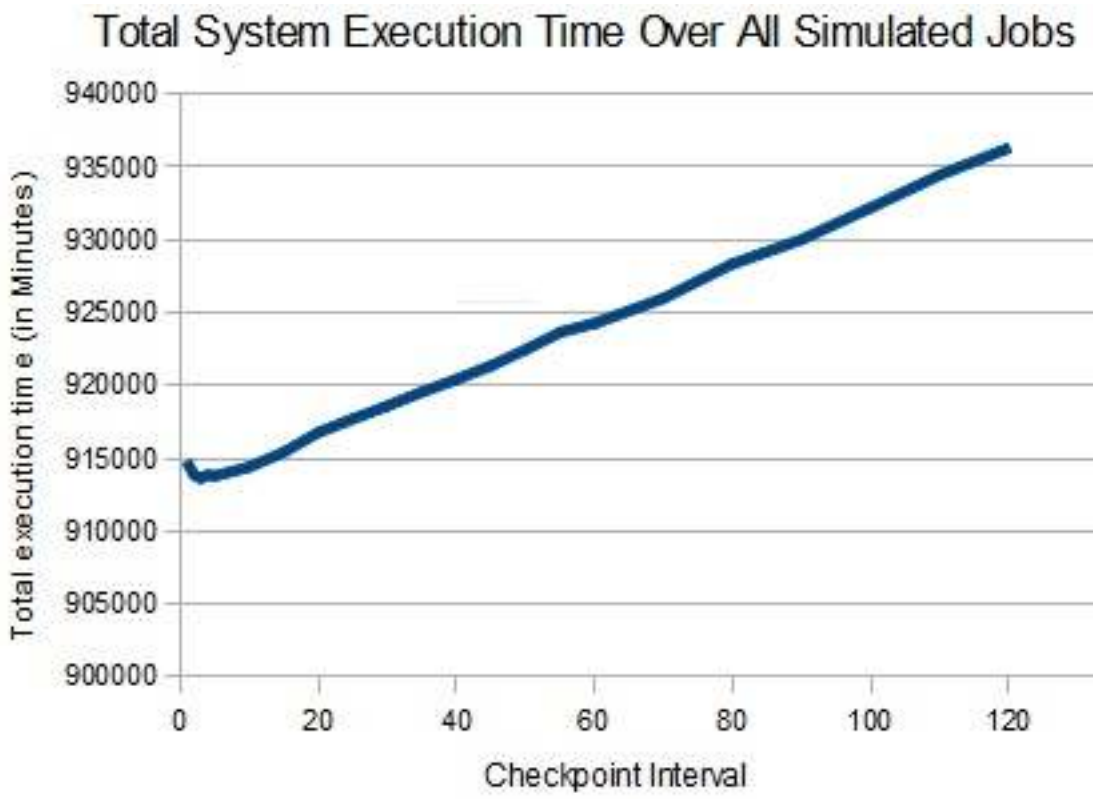

Figure 4. 7 Execution Time for All Jobs Showing the Impact of Checkpoint Interval on System Performance

The graphs depicting the average number of defensive I/O operations for systems 18 and 19, Figures 8 and 9, plot number of I/O operations as a function of checkpoint interval for simulations that used failures from LANL systems 19 and 18, respectively. Both graphs show that there is a clear reduction in I/O operations as the interval increases. For the simulations conducted, the values are a close match to the analytical model of Arunagiri, et al. (Arunagiri, Daly, & Teller, 2009), with the error being within one I/O operation. The graphs are so close that they are indistinguishable in these views. The data shows that the majority of the defensive I/O operations were simply the checkpoints conducted, which is easily predicted with solution time divided by checkpoint latency. Very few restarts were needed, averaging less than one restart per process for the simulations conducted.

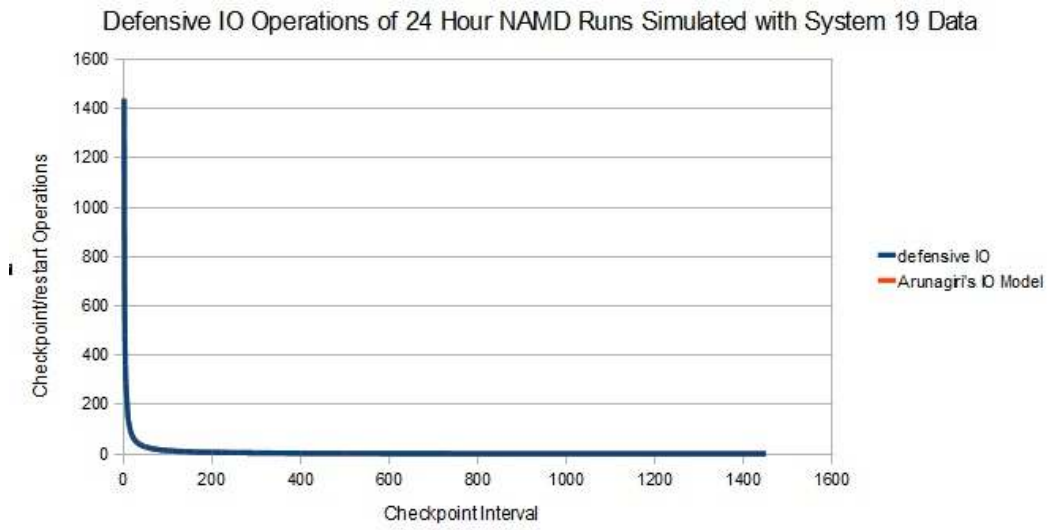

Figure 4. 8 System 19: Average Number of Defensive I/O Operations vs. Checkpoint Interval

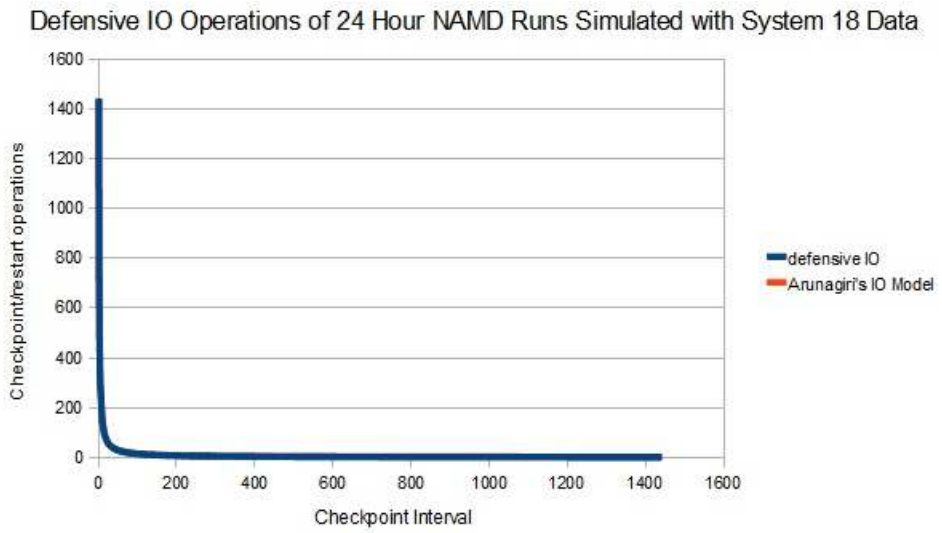

Figure 4. 9 System 18: Average Number of Defensive I/O Operations vs. Checkpoint Interval

These results indicate that the analytical models of Arunagiri (Arunagiri, Daly, & Teller, 2009) and Daly (Daly, 2006) can give very close approximations and still act as guidelines even on systems where the failure distribution does not explicitly match the exponential failure distribution assumptions. This information may not seem very useful since it is still challenging

to a priori estimate correct MTTI values and variances. To check how MTTI variance impacted the models, the values of MTTI were calculated for systems 18 and 19 at six-month intervals. The values of MTTI ranged from 368 to 530 days in these six-month computations of MTTI. For all the values, the optimal checkpoint still fell between three and four minutes for these NAMD runs. This suggests that the analytical models may still provide useful guidelines even with limited and varying MTTI data on systems where the failure distribution pattern is unknown.

4.7 E-M Clustering on LANL Data

For the clustering, Weka (Waikato Environment for Knowledge Analysis) (Hall, et al., 2009), a popular suite of machine-learning software written in Java, was used to find patterns in the LANL failure data from CFDR for the five systems that had the most reported failure events: systems 2, 16, 18, 19, and 20. These were chosen simply because they had a larger dataset to work with, allowing for more significant data analysis. Some had a low node MTTI, e.g., system 2, which had an average MTTI of less than two months. Others, like systems 18 and 19, had a more reasonable node MTTI, falling between one and two years. Due to the wide variety in system MTTI and hardware, it was important to analyze these systems independently of each other. 2-D graphs were produced using Weka, and 3-D graphs were produced using R (The Comprehensive R Archive Network) with GGobi (GGobi Data Visualization System).

4.7.1 Clustering

E-M clustering was conducted simply through use of Weka's E-M clustering algorithm using the default parameters. Each of the five chosen systems had E-M clustering conducted on them.

4.7.2 Results

The results of the E-M clustering for system 2, shown in Figure 4.10, indicate clusters of increased failures shortly after all the nodes went online and identified a group of three nodes that had a higher failure rate than surrounding nodes. System 16 showed no pattern as all failures were grouped into a single cluster and, therefore, the graph is not presented in this section. For systems 18 and 19 we were able to obtain data to identify the x-y positions and this was used to plot failures in Figures 4.11 and 4.13, respectively. As can be seen, the failures were grouped mostly on the geography of the node location. As shown in Figure 4.12, system 20 showed a pattern of increased failures when all 512 nodes were online, but when that early high failure data was excluded from the analysis, then the failures were grouped in a single cluster.

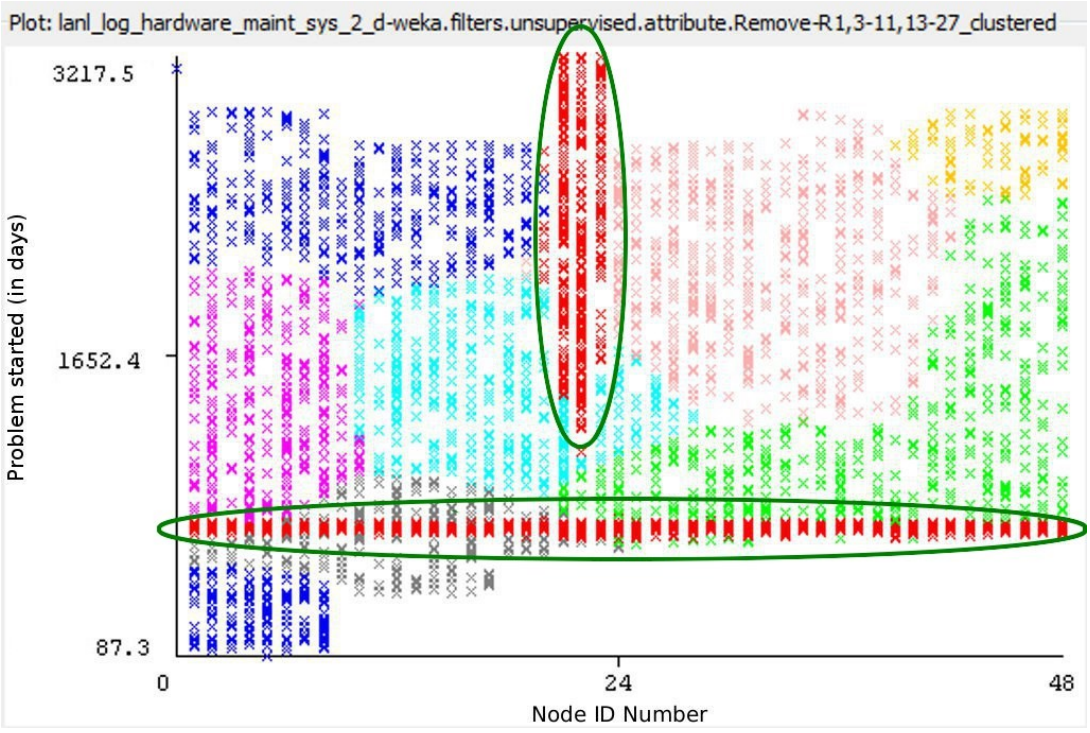

Figure 4.10 E-M Clustering on LANL System 2 showing Two High-Density Clusters

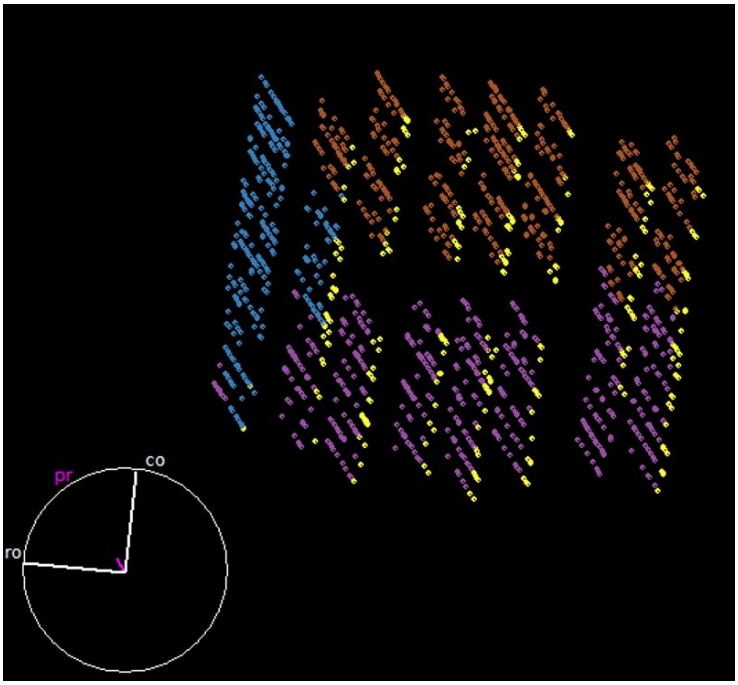

Figure 4.11 E-M Clustering on LANL System 18

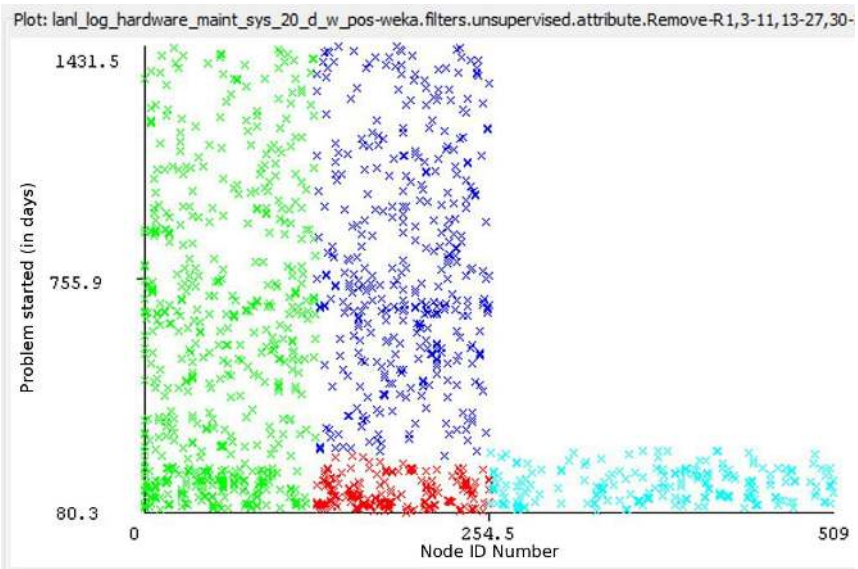

Figure 4.12 E-M Clustering on LANL System 20

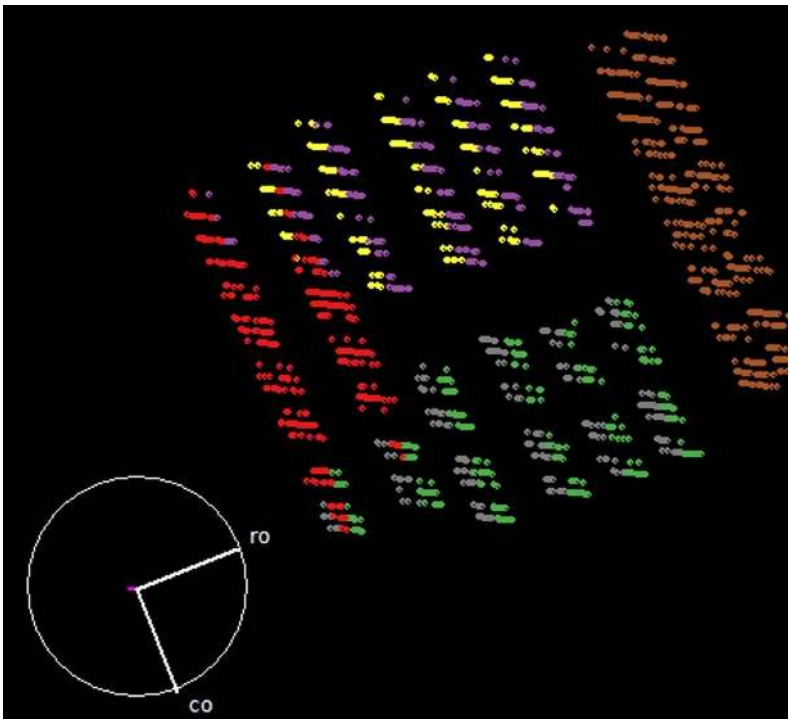

Figure 4.13 E-M Clustering On LANL System 19

4.8 K-means with Cross-Validation Clustering on LANL Data

Weka was again used to run K-Means clustering on the LANL system data.

4.8.1 Experiments

K-Means clustering was performed through a slightly more involved process. Simple K-Means in Weka requires the setting of the K-value in advance. To find the best fit, K-Means was

conducted for each system for K-values ranging from 10 to 200, incrementing at intervals of 10. The K-value used in the final clustering was determined based on which K-value generated the best log-likelihood value. Since log-likelihood is a common standard for comparing the fitness of a model for a specific dataset, the one with the best score was deemed to be the best clustering assignment.

4.8.2 Results

As seen in the corresponding graphs, the results of K-Means clustering show clear potential for temporal and spatio-temporal clustering. However, this only suggests the possibility of clusters. There is no guarantee that the clusters are not simply random. More information and testing are needed to determine the significance of the assigned clusters. The histogram in Figure 4.19 shows the time between failures and suggests that temporal clustering is likely as failures are more probable to occur close to a failure.

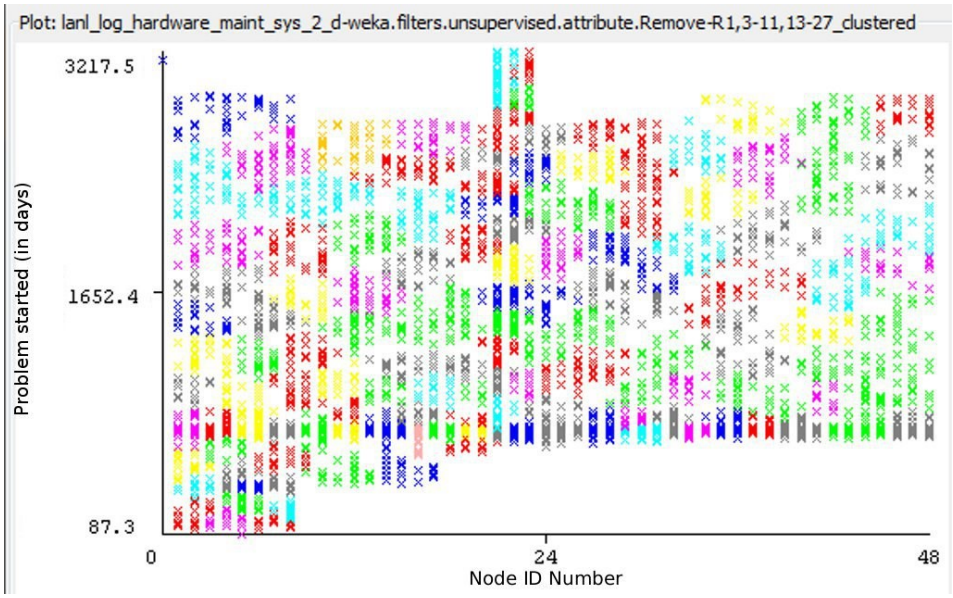

Figure 4.14 K-means with Density-based Clustering for System 2

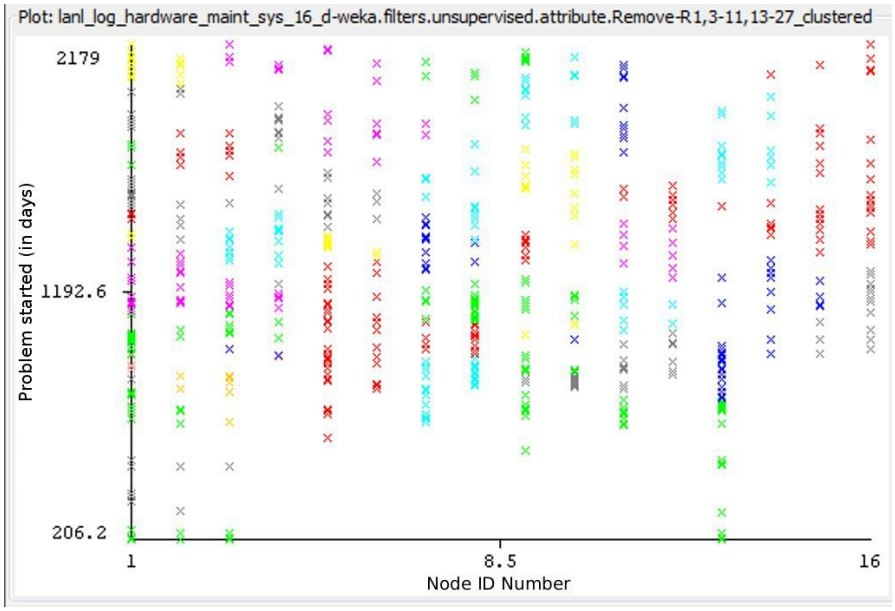

Figure 4.15 K-Means with Density-based Clustering for System 16

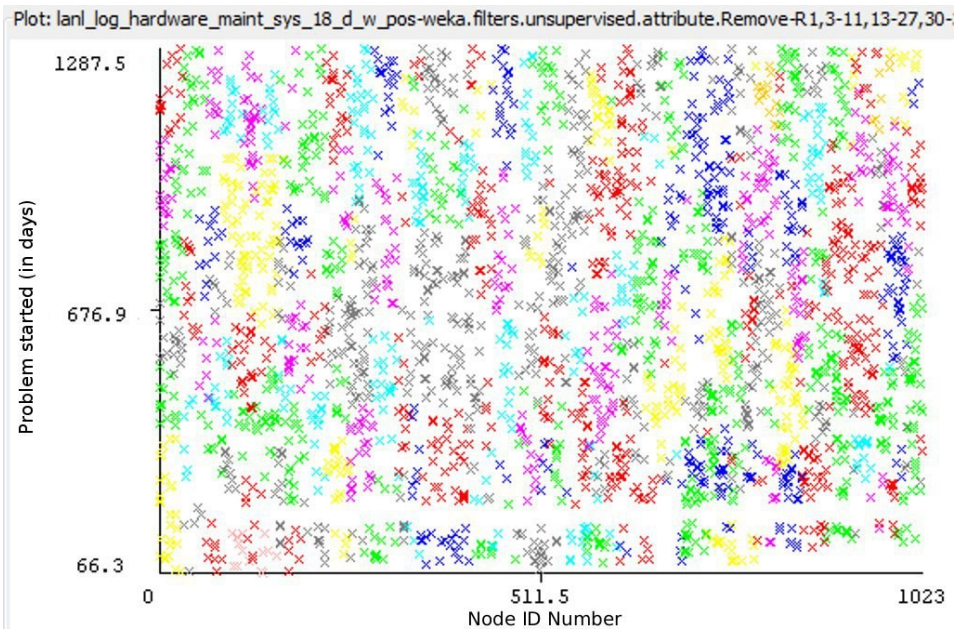

Figure 4.16 K-Means with Density -based Clustering for System 18

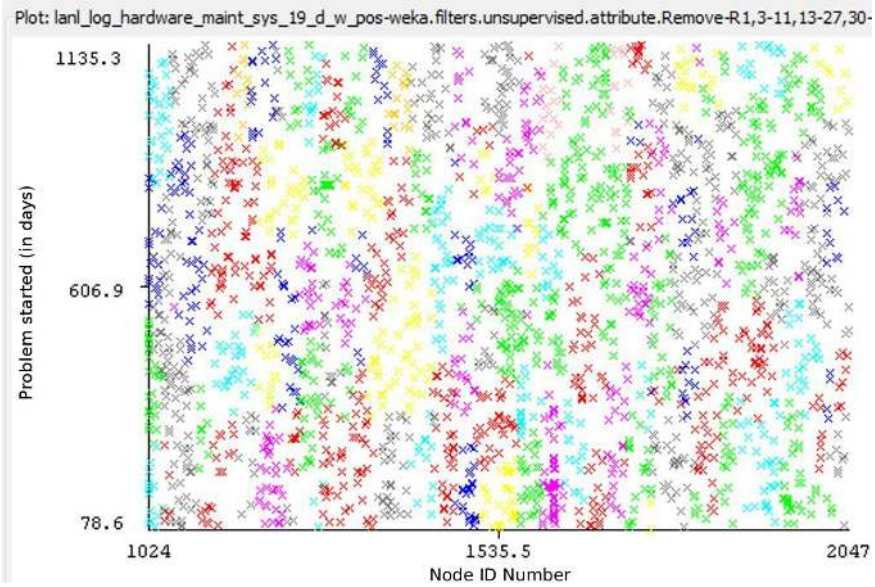

Figure 4.17 K-Means with Density-based Clustering for System 19

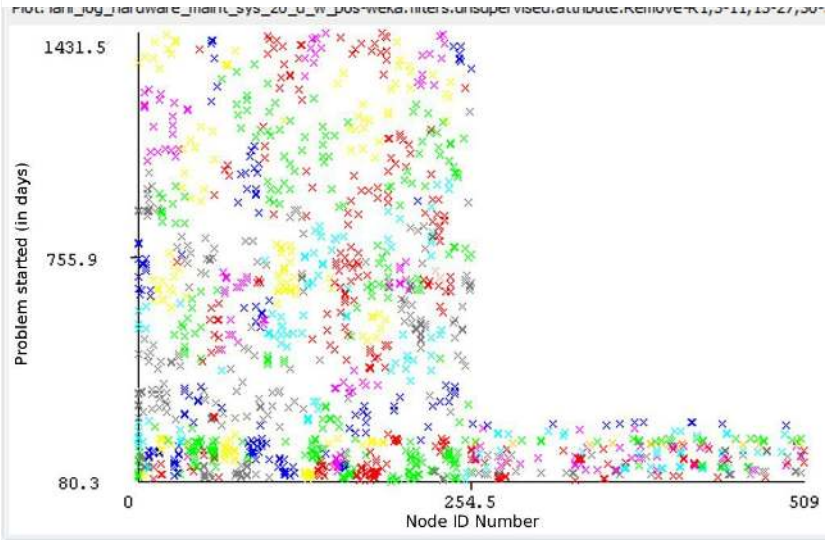

Figure 4.18 K-Means with Density-based Clustering for System 20

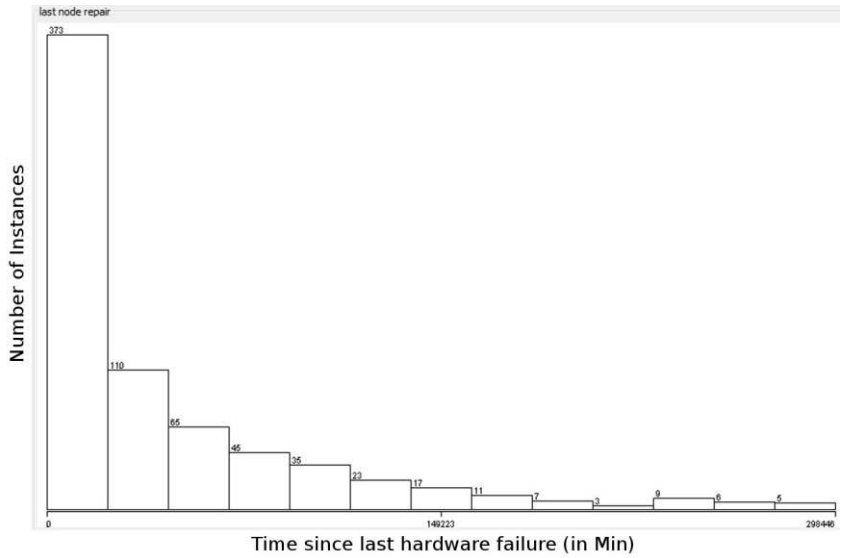

Figure 4. 19 Histogram of Number of Failures within a Given Time of Last Repair For LANL System 16.

5. Conclusions and Future Work

The conclusions drawn from this research are presented in Section 5.1. Section 5.2 concludes this thesis with related possible research directions.

5.1 Conclusions

Three main conclusions can be drawn from this work. The first is associated with the main objective of this thesis, i.e., using the community code NAMD executed on a high-performance computing system to validate the results of two analytical models associated with checkpointing. These two models predict the execution time and number of checkpoint I/O operations generated for periodic checkpointing applications, respectively. As discussed in Chapter 4, the number of trials, six per experiment, of NAMD on Ranger was not sufficient to accurately compare the resultant empirical checkpoint data with predictions made by analytical models based on inferential statistics. Nonetheless, using this empirical data we were able to investigate the behavior of the number of checkpoint I/O operations and the wall clock application execution time as a function of the checkpoint interval. The data showed that (1) for NAMD the number of checkpoint I/O operations decreased with increasing checkpoint interval, which is also the trend indicated by the analytical model of Arunagiri, et al. [4]; (2) for NAMD the number of NAMD timesteps needed for recomputation due to failures increased as the checkpoint interval increased; and (3) for NAMD the shape of the graph of execution time versus checkpoint interval does not look like a convex function with a single minimum and, therefore, does not explicitly match the behavior predicted by Daly's execution-time analytical model [3].

This could be because the number of trials is too small or because the values of the parameters used to compute the optimal checkpoint interval are not accurate.

The second conclusion is linked to the simulation study that was used to determine if these models can provide guidance with respect to selecting a proper checkpoint interval. This study investigated the efficacy of these models when the failure distribution is not an exponential distribution. As discussed in Chapter 4, checkpoint simulation results showed that even applications that only execute for a day can incur large recomputation costs over the lifetime of an HPC system if periodic checkpointing is not performed frequently enough. Accordingly, it would be of great benefit for HPC administrators to create checkpointing guidelines for users using programs that employ periodic checkpointing. Additionally, even though failures in large-scale HPC systems do not always show a propensity for exponential distribution, this research showed that, for the NAMD experiments conducted, that Daly's analytical model [3] for periodic checkpointing applications closely reflected both the execution time and the optimal checkpoint value for an application running on an HPC system. This research also showed that for NAMD using the analytical models of Daly and Arunagiri, et al. may provide a very good guideline for expected execution time and the number of checkpoint I/O operations even on systems that do not exhibit exponential failure rates. With this information, checkpoint intervals can be adjusted in order to significantly reduce defensive I/O operations while only slightly increasing checkpoint/restart overhead on systems where I/O contention is a source of performance issues.

And, finally, the third conclusion is related to the clustering study, the objective of which was to investigate if clustering of failure data could be used to help guide the initiation of checkpoint I/O. The E-M clustering of the failure data associated with LANL HPC systems identified areas of higher failures, such as LANL's system 2, where there were nodes that

exhibited more failures than others, and where there was a time when many more failures occurred across the entire system. However, this method did not reveal much information that might be useful in predicting failures as some systems studied did not exhibit any patterns at all in the E-M clustering. K-Means clustering may or may not prove useful in predicting failures. Further analysis, such as autocorrelative analysis, would be needed to determine if the clusters are significant or simply random.

5.2 Future Work

This section addresses the issues not addressed in this thesis that may show potential for future work. Three areas of continuing work, discussed in the following three subsections, are identified: (1) conducting spatio-temporal autocorrelation to determine to what degree failures in HPC systems cluster; (2) further work into identifying what is and what is not predictable in failure analysis; and (3) using this predictability information, develop better predictive algorithms to increase the efficiency of checkpointing.

5.2.1 Cluster Analysis: Spatio-Temporal Autocorrelation

Spatio-temporal autocorrelation is a means of determining the relative degree of correlation of clustered data. Because random data will sometimes cluster, observing clusters in data does not guarantee that the data is not random. Autocorrelative tests like Moran's I test for autocorrelation give indications of the degree of clustering and dispersion in a dataset. For example, Moran's I returns a value between -1 and 1. It will return a positive value for data that shows a tendency to cluster, 0 for data that is random, and a negative value for data that tends to disperse. Running appropriate autocorrelative tests on the studied failure data will give an

indication of whether the clusters we observed are comprised of related events or if they are simply random clusters of data.

5.2.2 Predicting Failures to Further Improve Checkpoint Performance

Exponential distribution of data is fundamentally stateless. If a component in a system has an exponential distribution, it will have a certain, unchanging probability of failing within a given time window. This means that the probability remains the same regardless of how long the system has been running. For example, if a component has a 70% chance of failing within one year at the start of its life, a year from then, if it has not failed, the probability that it will fail in the next year is still 70%. This makes failure prediction utterly impossible as the probability of failure within the next year is constantly 70%. Fortunately, as has been seen, some people have made progress in predicting failures.

Spatio-temporal clustering shows great promise in providing a means of predicting certain failures, which will allow preventative checkpointing to reduce computation loss on nodes where clustering makes failures more likely to occur. If we know what data is important to predicting failures, machine-learning algorithms could potentially be used to reduce the total execution time of applications that periodically checkpoint by reducing the total number of checkpoints needed. Some work has already been done to predict failures including using Bayesian networks, neural-gas clustering, and other machine-learning techniques [5] [6] [11] [7].

5.2.3 Determining Degree of I/O Contention and Tuning Checkpoints To Improve Network File System Performance

Ultimately, this research is aimed towards the goal of reducing I/O contention on network file systems, while not incurring additional computational costs. This research and the research of many others show the value of checkpointing and minimization of computation loss. In cases where I/O contention on a network file system is causing performance degradation, reducing the number of I/O operations by tuning checkpoint frequency could potentially improve overall system performance. The analytical models studied in this thesis could enable this.

Bibliography

Ansel, J., Arya, K., & Cooperman, G. (2009). DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. *23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS '09).*

Arunagiri, S., Daly, J. T., & Teller, P. (2009). Modeling and Analysis of Checkpoint I/O Operations. *ASMTA 2009.*

Barth, B., Brown, J., Hammond, J., & Phillips, J. (2012, 3 8). Meeting to Discuss Collaberation and Findings in Checkpoint Experiments.

Bhandarkar, M., Bhatele, A., Bhom, E., Brunner, R., Bueleus, F., Chipot, C., . . . al., e. (2010). *NAMD User's Guide Version 2.7b3.* Urbana, IL: Theoretical Biophysics Group University of Illinois and Beckman Institute.

Bougeret, M. (2011). Checkpointing Strategies for Parallel Jobs. *SC '11 Proceedings of the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.*

Browne, J., & Hammond, J. (2010, 4 9). Teleconference on Programs Run at TACC That Utilize Checkpointing.

Cappello, F. (2009). Fault Tollerance in Petascale/Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *The International Journal of High Performance Computning Applications, 23*(3), 211-226.

Chakraborty, B. (2012). Fault Tolerance: Validating a Mathematical Model via a Case Study of RAxML, an HPC Community Code. ETD Collection for University of Texas, El Paso.

Cluster Analysis - Wikipedia. (n.d.). Retrieved 2011, from http://en.wikipedia.org/wiki/Cluster_analysis

Daly, J. T. (2006). A Higher Order Estimate of the Optimum Checkpoint Interval for Restart Dumps. *Future Generation Computer Systems*.

DMTCP: Distributed MultiThreaded CheckPointing. (n.d.). Retrieved 2010, from http://dmtcp.sourceforge.net/

Exelixis Lab. (n.d.). Retrieved 2012, from http://www.exelixis-lab.org/

GGobi Data Visualization System. (n.d.). Retrieved 2011, from http://www.ggobi.org

Gu, J., Zheng, Z., & Lan, Z. (2008). Dynamic meta-learning for Failure Prediction in Large-scale Systems: A Case Study. *Proceedings of the International Conference on Parallel Processing.*

Gupta, M. R., & Chen, Y. (2010). Theory and Use of the EM Algorithml. *Foundation and Trends in Signal Processing, 4*(3), 223-296.

Hacker, T. J., Romero, F., & Carothers, C. D. (2009). An Analysis of Clustered Failures on Large Supercomputing Systems. *Journal of Parallel and Distributed Computing*.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explorations, 11*(1).

Jones, W. M., Daly, J. T., & DeBardeleban, N. (2010). Impact of Sub-optimal Checkpoint Interval on Application Efficiency in Computational Clusters. *HPDC '10.*

K-Means Clustering - Wikipedia. (n.d.). Retrieved 2011, from http://en.wikipedia.ord/wiki/K-means_clustering

Liang, Y., Zhang, Y., & Xiong, H. (2007). Failure Prediction in IBM BlueGene/L Event Logs. *7th IEEE Conference on Data Mining.*

Liu, Y., Nassar, R., Leangsuksun, C., Naksanehaboon, N., Paun, M., & Scott, S. (2007). A Reliability-Aware Approach for an Optimal Checkpoint Restart Model in HPC Environments . *IEEE International Conference on Cluster Computing.*

Mei, C., Sun, Y., Zheng, G., Bohm, E. J., Kale, L. V., Philips, J. C., & Harrison, C. (2011). Enabling and Scaling Biomolecular Simulations of 100 Million Atoms on Petascale Machines with a Multicore-optimized Message-driven Runtime. *SC '11.*

Moody, A. (2009). Overview of the Scalable Checkpoint Restart(SCR) Library. *Resilience Summit 2009.*

NAMD Utilities. (n.d.). Retrieved 2011, from http://www.ks.uiuc.edu/Research/namd/utilities/

Oliner, A., & Stearley, J. (2007). What Supercomputers Say: A Study of Five System Logs. *37th Annual International Conference on Dependable Systems and Networks.*

Parallel Programming Laboratory. (n.d.). Retrieved 2010, from http://charm.cs.uiuc.edu/charm/

Philips, J. C., Braun, R., Wang, W., Gumbart, J., Tajkhorshid, E., Villa, E., . . . Schulten, K. (2005). Scalable Molecular Dynamics with NAMD. *Journal of Computational Chemistry, 26*, 1781-1802.

Philips, J., Zheng, G., & Kale, L. (2002). Biomolecular Simulation on Thousands of Processors. *SC 2002*.

Phillips, J., & Hammond, J. (2010, 7 23). Teleconference on the Properties of NAMD and NAMD Checkpointing.

Scalable Checkpoint/Restart. (n.d.). Retrieved 2010, from http://souceforge.net/projects/scalablecr/

Schroeder, B., & Gibson, G. (2006). A Large-Scale Study of Failures in High-Performance Computing Systems. *International Symposium on Dependable Systems and Networks*.

Shastry, M., & Venkatesh, K. (2010). Analysis of Dependencies of Checkpoint Cost and Checkpoint Interval of Fault Tolerant MPI Applications. *International Journal on Computer Science and Engineering, 2*(8), 2690-2697.

Stamatakis, A., Aberer, A. J., Groll, C., Smith, S. A., Berger, S., & Izquierdo-Carrasco, F. (2012). RAxML-Light: A Tool for Computing TeraByte Phylogenies. *Bioinformatics*.

TCL Developer Site. (n.d.). Retrieved 2011, from http://www.tcl.tk/

Texas Advanced Computing Center Ranger-User-Guide. (n.d.). Retrieved 2010, from http://www.tacc.utexas.edu/user-services/user-guides/ranger-user-guide

The Comprehensive R Archive Network. (n.d.). Retrieved 2011, from http://cran.r-project.org

Usenix - Computer Failure Data Repository (CFDR). (n.d.). Retrieved 2011, from http://cfdr.usenix/org

Zheng, Z., Lan, Z., Gupta, R., Coghlan, S., & Beckman, P. (2010). A Practical Failure Prediction With Location and Lead Time for BlueGene/P. *Dependable Systems and Networks Workshop, DSN 2010.*

Zheng, Z., Lan, Z., Park, B. H., & Geist, A. (2009). System Log Pre-processing to Improve Failure Prediction.

Zheng, Z., Yu, L., Tang, W., & Zhiling, L. (2011). Co-analysis of RAS Log and Job Log on BlueGene/P. *IEEE International Parallel and Distributed Processing Symposium.*

Zhong, S., Khoshgoftaar, T. M., & Seliya, N. (2004). Analyzing Software Measurement Data with Clustering Techniques. *Intelligent Systems, IEEE, 19*(2), 20-27.

Appendix

A. Code modifications to NAMD

In order to collect data on the interval and latency of checkpointing in NAMD, functions were added to NAMD to monitor the generation of the NAMD's restart files. To add the monitoring, changes were made to the following files:

1. Output.h
    i. Added function declarations to object Output:
    ii. Private member function: get_time() which uses gettimeofday() to get the time expressed in seconds and miliseconds and returns the value as a float-point value.
    iii. Public member function: output_checkpoint_buffer() which outputs the checkpoint buffer to a file.
    iv. Public member function: log_checkpoint_event(int,long int,int) which adds an event entry into our checkpoint logging buffer.
    v. Added struct_event_log, the struct used by our checkpoint log buffer.
    vi. Added constants :
        1. BUFFER_SIZE: controls how large the buffer size will be.
        2. FLUSH_INTERVAL: controls how often contents of the buffer will be flushed to the file.
2. Output.C
    i. Added actual functions as described above.

ii. Added global variables checkpoint_buffer, checkpoint_buffer_count, checkpoint_log_name, start_time

iii. Added call to log_checkpoint_event() to mark the end of the restart velocities being written to file.

iv. Added call to get_time() in constructor for Output object so that the clock will be initialized at the beginning of execution.

3. Control.C

i. Added call to function Output::log_checkpoint_event() to mark the time before writing of the XCS restart file.

With these changes, we were tracking not only the interval that is represented by checkpointing at different times, but we also collect data about the latency produced by the checkpoint operation.

B. Failure Simulating TCL Script

Here is the full code of the TCL script.

```
# Fail lib is a collection of functions that simulate failures based on the values passed.

# dorun proc used to fix bit overflow time problem.  Returns execution time in miliseconds instead of micro.

proc dorun {runsteps splitsize} {
```

```
        #global splitsize;

        print "$runsteps/$splitsize";

        set itterations [expr {$runsteps/$splitsize}]; # integer division to make things
nice.

        set remainder [expr {$runsteps%$splitsize}]; # we need the remainder to run all
the steps

        set floatmilitime 0;

        if {$itterations > 0} {

         # run for stepsize steps.

         set tempstrings [split [time {run $splitsize} $itterations]];

         set temptime [lindex $tempstrings 0];

         set floatmilitime [expr {($temptime/1000.0)*$itterations}]; # avg * itterations =
total run time

        }

        if {$remainder> 0} {

         set tempstrings [split [time {run $remainder}]];

         set temptime [lindex $tempstrings 0];
```

```
 set floatmilitime [expr {$floatmilitime+($temptime/1000.0)}];

 }

 set intmilitime [expr {int($floatmilitime)}]

 return $intmilitime

}

# This is a recursive function that treats a list like a heap.  It assumes the list is already heaped.

# A sorted array/tcl list has the property of also being in heap order.

# It takes a value and inserts it at the current node if the value is <= its children

# or bubbles up the lower of the children and recursively calls on that child's node.

#proc gettime {} {

#return [clock seconds];

#}

proc ReplaceAndReheap {mylist listindex newvalue} {

#set tracefile [open trace.txt a]

set leftchild [expr {2 * $listindex +1}]; # this is the standard for array represented heaps where the root node = 0;
```

```
set rightchild [expr {$leftchild+1}];

set listlength [llength $mylist];

#puts $tracefile "$listlength elements: $listindex node: $newvalue value: left $leftchild: right $rightchild";

#flush $tracefile;

#close $tracefile;

# below condition screens for possibility when there are no children, so we are done.

if {$leftchild>=$listlength} {

lset mylist $listindex $newvalue;

return $mylist;

} else {

#another special condition below screens for when there is only a left child.

if {$rightchild>=$listlength} {

if {[lindex $mylist $leftchild]<$newvalue} {

# less than means left child value bubbles up and is replaced with newvalue

lset mylist $listindex [lindex $mylist $leftchild];
```

```
            lset mylist $leftchild $newvalue;
              } else {
               lset mylist $listindex $newvalue;
              }
              return $mylist;  # if we have no right child, we have reached the end of the heap, we are done.
            } else {
              if {($newvalue<[lindex $mylist $leftchild]) && ($newvalue<[lindex $mylist $rightchild])} {
               # we're done set the newvalue here and return the list
               lset mylist $listindex $newvalue;
               return $mylist;
              } else {
               # smaller child bubbles up, we then recursively call on that node.
               if {[lindex $mylist $leftchild]<[lindex $mylist $rightchild]} {
             lset mylist $listindex [lindex $mylist $leftchild];
             set mylist [ReplaceAndReheap $mylist $leftchild $newvalue];
```

```
            return $mylist;
        } else {
         lset mylist $listindex [lindex $mylist $rightchild];
         set mylist [ReplaceAndReheap $mylist $rightchild $newvalue];
            return $mylist;
        }
          }
         }
         }
         return {0};
        }
        # End of heap functions.  That's it, no more needed for this application.
        # Main function.  Called to run a certain number of steps with the given failure properties.
        proc runwithfailures {nodes mtti totaltimesteps expectedtime restartfreq restartname splitsize} {
         # setting output
```

```
set logfile [open output.txt a]



puts $logfile "Output begins.";

#::tcl::tm::path add {"C:\tcl\lib\libtcl-1.12"};

#puts $logfile [::tcl::tm::path list];

#flush $logfile;

#lappend auto_path {c:\tcl\lib\tcllib-1.12};

#puts $logfile $auto_path;

#package require math::statistics;  # needed for random numbers with exponential distribution.

#output config values to log file so that we know what the data is for.

puts $logfile "Expected execution time: $expectedtime";

puts $logfile "MTTI: $mtti";

puts $logfile "Total timesteps: $totaltimesteps";

puts $logfile "Restart frequency: $restartfreq";

puts $logfile "Nodes: $nodes";

# Computing values needed for main computation
```

```
set timestep 0;

set randfilename "randoms_$mtti.txt";

set randomfile [open $randfilename r];

gets $randomfile randomstring;

set erandoms [split $randomstring " "];

set randomcount [llength $erandoms];

close $randomfile;

set failurelistraw [lindex $erandoms 0];

for {set i 1} {$i<$nodes} {incr i} {

  lappend failurelistraw [lindex $erandoms $i];  # loading the first p numbers into the failure time heap

}

set rindex $nodes;

set failureheap [lsort -real $failurelistraw];

puts $logfile "Test1";

flush $logfile;

set lastcheckpoint 0;
```

```
checkpoint;

set firstfailurestep [expr {int ([lindex $failureheap 0]*60*3)}]; #estimate on three seconds per step

set timetest 110; # setting a reasonable expected execution step # to get a good average to work off of.

while {$timetest > $firstfailurestep} {

  incr timetest -10;  #making sure the number is less than the time of the expected first failure

}

incr timetest -10;  #adding a buffer in case we overestimated the time per step.

#run namd a given number of time steps less than first expected failure to estimate run time/time step

print $timetest;

set mtime [dorun $timetest $splitsize];

set lastcheckpoint [expr {$timetest-$timetest%$restartfreq}];

set trueruntime [expr {$mtime/1000.0}];

set truerunsteps $timetest;

set failurecount 0;
```

```
set timestep $timetest

print "timeperstep= trueruntime/$truerunsteps)/60";  # in minutes

set timeperstep [expr {$trueruntime/$truerunsteps/60}];

print "$timeperstep= trueruntime/$truerunsteps)/60";

while {$timestep<$totaltimesteps} {

 # estimate how many time steps to execute before next failure based on
execution time/timesteps estimate.

 set nextfailuretime [lindex $failureheap 0];

 set nextfailurestep [expr {int ($nextfailuretime/$timeperstep)}];

 print "$nextfailurestep=$nextfailuretime/$timeperstep";

 # check if next failure step < totalsteps we run to completion then terminate if
totalsteps is less

 if {$nextfailurestep>$totaltimesteps} {

  set runsteps [expr {$totaltimesteps-$timestep}];

  incr truerunsteps $runsteps;

  set mtime [dorun $runsteps $splitsize];

  set trueruntime [expr {$trueruntime+($mtime/1000.0)}];
```

```
    set timestep $totaltimesteps;
   } else {
    incr failurecount;
    # changing things around, run until next checkpoint, perform checkpoint, run until failure, then revert.
    set nextcheckpoint [expr {$nextfailurestep-$nextfailurestep%$restartfreq}]
    set runsteps [expr {$nextcheckpoint-$timestep}];
    print "runsteps $runsteps; last $lastcheckpoint; next $nextcheckpoint."
    if {$runsteps>0} {
     set mtime [dorun $runsteps $splitsize];
     set trueruntime [expr {$trueruntime+($mtime/1000.0)}];
     incr truerunsteps $runsteps;
     checkpoint;
    }
    set runsteps [expr {$nextfailurestep-$nextcheckpoint+5}];
    set runsteps [expr {$runsteps-($runsteps%10)}]; #Due to NAMD requirements, steps need to be multiples of 10
```

```
        # run estimated number of steps to execute until next failure time.

        print "run steps: $runsteps; nextfailurestep: $nextfailurestep; lastcheckpoint:
$lastcheckpoint.";

        set mtime [dorun $runsteps $splitsize];

        set trueruntime [expr {$trueruntime+($mtime/1000.0)}];

        set lastcheckpoint $nextcheckpoint;

        incr truerunsteps $runsteps;

        # Reload coordinate and velocities file from last checkpoint files.

        set runtime $trueruntime;

        puts $logfile "Failure occurred at time step $nextfailurestep and run time
$runtime.";

        print "Failure occurred at time step $nextfailurestep and run time $runtime.";

        set timestring [split [time {

          revert

          set coorfilename "$restartname.coor";

          set velfilename "$restartname.vel";

          set coors [open $coorfilename r];
```

```
  set vels [open $velfilename r];
  set junkcoors [read $coors];
  set junkvels [read $vels];
 }]]
 set junkcoors "";
 set junkvels "";
 set mtime [lindex $timestring 0]
 set trueruntime [expr {$trueruntime+($mtime/1000000)}];
 puts $logfile "System restored to point at $lastcheckpoint at run time
$trueruntime.";
 print "System restored to point at $lastcheckpoint at run time $trueruntime.";


 # pull next erandom, add it to the root value and then replace and reheap.
 if {$rindex>=$randomcount} {
  #this shouldn't happen often.  We need to generate more random values
initially if this does happen. Print an error and reset rindex to 0.
```

```
                puts $logfile "Error: Ran out of random numbers, had to reset rindex.  Please
generate a larger number of random numbers.";
                set rindex 0;
              }
              set nextrandom [lindex $erandoms $rindex];
              set replacementfailuretime [expr {$nextfailuretime+$nextrandom}];
              incr rindex;
              set failureheap [ReplaceAndReheap $failureheap 0 $replacementfailuretime];
              # Use total execution time thusfar and truetimesteps to improve time/step
estimate.
              set timeperstep [expr {($trueruntime/$truerunsteps)/60}];
              # Uncomment previous line in real test.
              set timestep $lastcheckpoint;
            }
          }
          #set totaltime [expr {$truerunsteps*$timeperstep}];
          set totaltime $trueruntime;
```

```
puts $logfile "Total time: $totaltime";

puts $logfile "Total steps executed: $truerunsteps";

puts $logfile "Total failures: $failurecount";

puts $logfile "Simulation completed successfully.";

# Main computation ends

flush $logfile;

close $logfile;

}
```

C. HPC Queue Simulator Code

This is comprised of two files: linked_list.java and Main.java

File linked_list.java:

```
/*

 * To change this template, choose Tools | Templates

 * and open the template in the editor.

 */
```

```
package checkpoint_simulator_w_n_n;



/**

 *

 * @author Michael

 */

class lnode

{



    public double time;

    public int job;

    public int run;

    public int node_count;

    public int[] nodes;

    public lnode next;
```

```
    public lnode(double t_val,int j_num, int r_num, int n_count, int[]n_array)
    {
       time=t_val;
       job=j_num;
       run=r_num;
       node_count=n_count;
       nodes=new int[n_count];
       for (int i=0;i<n_count;i++)
       {
          nodes[i]=n_array[i];
       }
       next=null;
       //return this;
    }
}

class linked_list
```

```
{
    public lnode first;
    private lnode pointer;
    private lnode p_parent;
    public linked_list()
    {
        first=null;
        pointer=first;
    }
    public linked_list(double time, int job, int run, int node_count, int[]nodes) // our
constructor for a new linked list
    {
        first=new lnode(time, job, run, node_count, nodes);
        pointer=first;
        p_parent=null;
        //this.time= value;
    }
```

```
public boolean isEmpty()

{

   return (first==null);

}
```

```
public lnode return_first() // goes through the list and returns the next value, pointing pointer to the next node.

{

   p_parent=null;

   pointer=first;

   return pointer;

}
```

```
public lnode return_next() // goes through the list and returns the next value, pointing pointer to the next node.

{
```

```
        if (pointer==null)
        {
            return pointer;
        }
        else
        {
            p_parent=pointer;
            pointer=pointer.next;
            return pointer;

        }
    }
```

public lnode return_current() // goes through the list and returns the next value, pointing pointer to the next node.

```
    {
        return pointer;
```

```
}


public void remove_node()
{
   if (!this.isEmpty())
   {
      if (p_parent==null)// special case, we are at the first node.
      {
         first=pointer.next;
         pointer=first;
      }
      else
      {
         if (pointer!=null)
         {
            p_parent.next=pointer.next;
            pointer=pointer.next;
```

```
            }
        }
    }

}

public void reset()
{
    pointer=first;
    p_parent=null;
}

public void push(double time, int job, int run, int node_count, int[]nodes)
{
    lnode new_node= new lnode(time, job, run, node_count, nodes);
    //System.out.println(value);
    if (this.isEmpty())
```

```
{
   first=new_node;
}
else
{
   //first case: the value is less than the smallest item in the list
   if (time<first.time)
   {
      new_node.next=first;
      first=new_node;
   }
   else // find the place it belongs
   {
      lnode parent_node=first;
      lnode current_node=parent_node.next;
      while ((current_node!=null)&&(current_node.time<time))
      {
```

```
                    //System.out.println(current_node.time);
                    parent_node=current_node;
                    current_node=parent_node.next;
                }
                if ((current_node==null)||(current_node.time>time)) // we don't want
duplicates
                {
                    parent_node.next=new_node;
                    new_node.next=current_node;
                }
            }
        }
        this.reset();
    }
    public lnode pop ()
    {
        if (this.isEmpty())
```

```
            {
               return null;
            }
            lnode return_value=first;
            first=first.next;
            this.reset();
            return return_value;
        }
    }
```

File Main.java:

```
    /*
     * To change this template, choose Tools | Templates
     * and open the template in the editor.
     */
    package checkpoint_simulator_w_n_n;
    /**
```

```
 *
 * @author Michael
 */
import java.util.Random;
import java.io.*;
//import java.lang.Math;
class parameters
{
    // Not really a class expected to be used for a vairable, just to store global
constants
    // change these to set the parameters for the simulation if you want to change
the
    // input files, output file, number of nodes, time to start the simulation from, etc.
    //
    // job_file_number: not really a parameter, just used in the filenames I made
    static int job_file_number=5;
    // job_file_name: name of the file from which to pull jobs.
```

```
        // file must use the format: #,#,#,#,#,# decimal values allowed
        // values may be comma or white-space separated
        // lines not matching the pattern will be ignored
        public static String job_file_name="jobs_"+job_file_number+"_f.csv";
        // failure_data_file: name of the file from which to pull failure information
        // file must use format: #,#,#,#,#
        // lines not matching the pattern will be ignored
        // allows failure data from multiple systems to be in the file so you don't
        // have to separate out the data you need or change the file to change the
system.
        // It will only use the data from the system indicated by system_number set
below
        public static String failure_data_file="failure_times.csv";
        // where to put the output.  Output is in CSV format.
        public static String output_file="output_"+job_file_number+".csv";
        // system_number: Failure data you want to use
        public static int system_number=18;
```

```
    // node_count: set to the number of nodes the system has.
    // If desired, this can be set to fewer or more nodes.
    // If the number of nodes is higher than the failure data has, those nodes
    // will never encounter a failure.  If it is lower, the failures will be
    // based only on the first n nodes.
    public static int node_count=1024;


    // start_time: This sets the time to the point in the failure data that
    // the simulation should start.  Since many of the LANL systems do not have
    // records of failures after install, only after production, this can be
    // used to start from a point where failures are being recorded.
    public static double start_time=129600;
}
class pq_node
{
    public double priority;
    // Insert everything that it needs to be here.
```

```
public int index;

public int event; // for event queue only.

public int run; // for events related to a running job only.

pq_node()

{

   priority=0;

   index=0;

   event=0;

   run=0;

}

pq_node(int new_index)

{

   priority=0;

   index=new_index;

   event=0;

   run=0;

}
```

```
    pq_node(int new_index, double new_priority)
    {
        priority=new_priority;
        index=new_index;
        event=0;
        run=0;
    }
    void copy_node(pq_node origin)
    {
        priority=origin.priority;
        index=origin.index;
        event=origin.event;
        run=origin.run;
    }
}
// The priority queue will be only indexes for an array.
// It's easier to manage things that way since there will be less copying.
```

```
// Since memory is less of a concern than computation time, this will be more
efficient.

class priority_queue // array implementation of heap type priority queue.

{

   public static final int DEFAULT_SIZE = 511;

   // DEFAULT SIZE defines the starting size of the heap.  It should be not so big
that

   // the queue takes much more memory than needed and not so small as to cause
frequent

   // resizing as that requires copying the existing contents of the queue to a larger
array.

   // 511 elements with 2 two int values should only take about 4 KB.

   int size;

   pq_node[] heap_array;

   int last;

   void init()

   {

      for(int i=0;i<size;i++)
```

```
        heap_array[i]=new pq_node();
}
priority_queue(int qsize)
{
   size=qsize;
   heap_array=new pq_node[size];
   last=-1;
   init();
}
priority_queue()
{
   size=DEFAULT_SIZE;
   heap_array=new pq_node[size];
   last=-1;
   init();
}
priority_queue make_copy()
```

```
{
   priority_queue copy_queue=new priority_queue(size);
   copy_queue.last=last;
   for (int i=0;i<=last;i++)
   {
      copy_queue.heap_array[i].copy_node(heap_array[i]);
   }
   return copy_queue;
}
void swap(int node1,int node2)
{
   pq_node temp= new pq_node();
   temp.copy_node(heap_array[node1]);
   heap_array[node1].copy_node(heap_array[node2]);
   heap_array[node2].copy_node(temp);
}
```

```
void resize (int new_size) //use sparingly as it must copy all the elements from the old array.
{
   pq_node[] new_heap=new pq_node[new_size];
   for(int i=0;i<new_size;i++)
      new_heap[i]=new pq_node();
   int copy_size=last;
   if (copy_size>new_size) copy_size=new_size;
   for (int i=0;i<copy_size;i++)
   {
      new_heap[i].copy_node(heap_array[i]);
   }
   heap_array=new_heap;
   size=new_size;
   if (last>size) last=size-1;
}
int bubble_up(int start_node)
```

```
{
   int parent_node=(start_node-1)/2;
   if
((start_node>0)&&(heap_array[start_node].priority<heap_array[parent_node].priority))
   {
      swap(parent_node,start_node);
      if (parent_node>0)
         return bubble_up(parent_node);
   }
   return start_node; // returns the node that the value eventually ends up in.
}
int push(int new_index, double new_priority, int event, int run)
{
   last++;
   if (last==size)
   {
      resize(2*size+1); // double the size
```

```
    }
    // insert into last position then bubble up
    heap_array[last].index=new_index;
    heap_array[last].priority=new_priority;
    heap_array[last].event=event;
    heap_array[last].run=run;
    return bubble_up(last);
}
int push(int new_index, double new_priority, int event)
{
    return push(new_index,new_priority,event,0);
}
int push(int new_index, double new_priority)
{
    return push(new_index,new_priority,0,0);
}
int trickle_down(int node)
```

```
{
   int right,left,lower;
   left=2*node+1;
   right=2*node+2;
   if (left<=last) // no left means no children, and were done
   {
      if (right <=last) // verify that the right child exists.
      {
         // both exist.  Find the lower.
         if (heap_array[left].priority<heap_array[right].priority)
            lower=left;
         else
            lower=right;
      }
      else // only the right child exists.  Swap or stop and we are done.
      {
         lower=left;
```

```
        }
        if (heap_array[lower].priority<heap_array[node].priority)
        {
            swap(lower,node);
            return trickle_down(lower);
        }
    }
    return node; // return node that the value ends up in
}
pq_node pop ()
{
    if (last==-1) return null; // the heap is empty
    else
    {
        pq_node return_node=new pq_node();
        return_node.copy_node(heap_array[0]);
```

```
        if (last>0)
        {
            heap_array[0].copy_node(heap_array[last]);
            trickle_down(0);
        }
        last--;
        return return_node;
    }
}
int increment(int node,double value)
{
    heap_array[node].priority+=value;
    if (value<0)// value was decreased.  Bubble up.
    {
        return bubble_up(node);
    }
    else
```

```
            {
                return trickle_down(node);
            }
        }
        void increment_all(double value)
        {
            for(int i=0;i<=last;i++)
                heap_array[i].priority+=value; // all are being updated, so no bubble or
trickle
        }
        void populate_event_queue()
        {
            // to be populated with code that fills the queue with the events.
            // This function is proprietary, it is only to be used with the
            // event heap.  Don't use for other heaps.
        }
    }
```

```
class jobs
{
   public int size;
   public static int job_file_number = 5;
   public job[] all_jobs;
   jobs()
   {
      size=0;
      // read all jobs from a csv file
      // need to count, reset, then populate.
      try
      {
         String filename=parameters.job_file_name;
         //System.out.println(filename);
         FileInputStream fstream = new FileInputStream(filename);
         // Get the object of DataInputStream
         DataInputStream in = new DataInputStream(fstream);
```

```
                BufferedReader br = new BufferedReader(new InputStreamReader(in));
                String strLine;
                String[] values;
                String num_pattern="-?[0-9]+(\\.[0-9]*)?";
                String
valid_pattern="\\s*"+num_pattern+"\\s*(((\\s)|(,\\s*))"+num_pattern+"\\s*){5,}?";
                while ((strLine = br.readLine()) != null)
                {
                   // Print the content on the console
                   values=strLine.split("\\s");
                   if ((strLine.matches(valid_pattern)))
                   {
                      size++;
                   }
                }
                System.out.println("Total valid jobs:"+size);
                fstream.close();
```

```
in.close();

br.close();

fstream = new FileInputStream(filename);

// Get the object of DataInputStream

in = new DataInputStream(fstream);

br = new BufferedReader(new InputStreamReader(in));

all_jobs=new job[size];

int count=0;

while ((strLine = br.readLine()) != null)

{

   //System.out.println("Count: "+count);

   values=strLine.split("((\\s)|(,\\s*))");

   if ((strLine.matches(valid_pattern)))

   {

      double arrival_time=Double.parseDouble(values[0]);

      double interval=Double.parseDouble(values[1]);

      double latency=Double.parseDouble(values[2]);
```

```
                    double solution_time=Double.parseDouble(values[3]);

                    int nodes_needed=Integer.parseInt(values[4]);

                    double restart=Double.parseDouble(values[5]);

                    // job(double a_time, double s_time, double i, double l, int n_needed)

                    all_jobs[count]=new job( arrival_time, solution_time, interval,
latency, nodes_needed, restart);

                    count++;

                }

            }

            fstream.close();

            in.close();

            br.close();

        }

        catch(Exception e)

        {//Catch exception if any

            System.err.println("Error: " + e.getMessage());

        }
```

```
    }
}
class job
{
    public double arrival_time;
    public double interval;
    public double latency;
    public double solution_time;
    public int nodes_needed;
    public double restart;
    public double execution_time;
    public double last_checkpoint;
    public double temp_value;
    public double next_checkpoint;
    public int [] nodes;
    public double start_time;
    public int failures;
```

```
public int checkpoint_writes;

job()

{

   arrival_time=0;

   interval=0;

   latency=0;

   solution_time=0;

   nodes_needed=0;

   execution_time=0;

   last_checkpoint=0;

   failures=0;

   checkpoint_writes=0;

}

job(double a_time, double s_time, double i, double l, int n_needed, double r)

{

   arrival_time=a_time;

   interval=i;
```

```
        latency=l;
        solution_time=s_time;
        nodes_needed=n_needed;
        restart=r;
        execution_time=0;
        last_checkpoint=0;
        failures=0;
        checkpoint_writes=0;
        nodes=new int[n_needed];
        for (int j=0;j<n_needed;j++)
        {
           nodes[j]=-1;
        }
    }
}
// Scheduler
// scheduler needs to do the following:
```

```
/* Scheduler
 * Does the following:
 * Check available resources to see if highest priority job will be able to run.
 * If it can, start the job, set first checkpoint and completion events for that job.
 * If not, compute when the needed resources will become available.
 * Run lower priority job if it will complete before the highest job can run.
 *
 * Problems: need to be able to update priority/time of jobs and easily track
 * when a job related event changes possition in the queue.
 * When Swapping, we need to update this somehow.
 *
 * Resources available and resource list should be available to the scheduler,
 * so should be part of it.
 * Should jobs be part of it as well?  Jobs are needed for creating start entries
 * and next checkpoint entries.  They will be needed for every checkpoint event.
 * How often will jobs need to be fed into it?  Jobs should probably be part of the
 * scheduler as well.
```

```
 *

 *

 */

class scheduler

{

    public static int system=parameters.system_number;

    public static int NODE_COUNT = parameters.node_count;

    public static final double MTTI = 500000; // node mtti

    public jobs task_list;

    public priority_queue schedule_heap;

    public int nodes_available;

    public boolean[] free_resources; // need to track what nodes are free or
occupied.

    public int[] node_jobs; // track which nodes have jobs running on them.

    public linked_list resources_freeing; // tracks when resources will become
available.

    public priority_queue event_heap;
```

```
public int f_size [];

public double failures[][];

public double repairs[][];

public int f_index[];

public int finished;

public double last_add;

scheduler(double start_time)

{

   init(start_time);

}

scheduler()

{

   init(-1.0);

}

void init(double start_time)

{

   task_list=new jobs();
```

```
schedule_heap=new priority_queue();

event_heap=new priority_queue();

nodes_available=NODE_COUNT;

node_jobs=new int [NODE_COUNT];

free_resources=new boolean [NODE_COUNT];

resources_freeing=new linked_list();

finished=0;

last_add=0;

//System.out.println("test2");

f_size= new int [NODE_COUNT];

f_index= new int [NODE_COUNT];

for (int i=0;i<NODE_COUNT;i++)

{

   f_size[i]= 0;

   f_index[i]=0;

}

Random generator=new Random();
```

```
failures=new double[NODE_COUNT][200];

repairs=new double[NODE_COUNT][200];

// Placeholder for now.  Change after initial testing is completed.

// <------------------------------------------------------------

// change to read from file:

// format: node,double\n

//         node,double\n...

try

{

   FileInputStream fstream = new
FileInputStream(parameters.failure_data_file);

   // Get the object of DataInputStream

   DataInputStream in = new DataInputStream(fstream);

   BufferedReader br = new BufferedReader(new InputStreamReader(in));

   String strLine;



   int j=0;
```

```
int node_num;
String unparsed[];
double fail_time;
double repair_time;
//System.out.println("test 0a");
//Read File Line By Line
while ((strLine = br.readLine()) != null)
{
   unparsed=strLine.split(",");
   node_num=Integer.parseInt(unparsed[1]);
   fail_time=Double.parseDouble(unparsed[2]);
   repair_time=Double.parseDouble(unparsed[3]);
   if
((system==Integer.parseInt(unparsed[0]))&&(fail_time>start_time)&&(node_num>=0)&
&(node_num<NODE_COUNT))
   {
      failures[node_num][f_size[node_num]]=fail_time-start_time;
```

```
            repairs[node_num][f_size[node_num]]=repair_time;
            f_size[node_num]++;
        }
    }
    br.close();
    in.close();
    fstream.close();
}
catch(Exception e)
{//Catch exception if any
    System.err.println("Error: " + e.getMessage());
}
// <-------------------------------------------------------------
// <-------------------------------------------------------------
//System.out.println("test3");
for (int i=0;i<NODE_COUNT;i++)
{
```

```
        node_jobs[i]=-1; // this may make free_resources redundant.  Wait and see.
        free_resources[i]=true;
        // schedule node failures
        if (f_size[i]>0)
            event_heap.push(i,next_failure(i,0),3);
    }
    //System.out.println("test4");
}
double next_failure(int node_num, double now) //
{
    double return_value=-1;
    while ((f_index[node_num]<f_size[node_num])&&(return_value<now))
    {
        return_value=failures[node_num][f_index[node_num]];
        if (return_value<now)
        {
```

```
                    System.err.println("Node "+node_num+" next failure time
"+f_index[node_num]+" was less than indicated repair time for previous failure.");

                    System.err.println("Last failure time:
"+failures[node_num][f_index[node_num]-1]+"; Last repair time:
"+repairs[node_num][f_index[node_num]-1]+"; This failure time: "+return_value);

                }

                f_index[node_num]++;

            }

            return return_value;

        }

        void run_job(int job_id,double time)

        {

            // populate nodes, update resources, create event (checkpoint or completion)

            // then create resource freeing item.

            int count=0;

            int i=0;
```

```
            while
((i<NODE_COUNT)&&(count<task_list.all_jobs[job_id].nodes_needed))
            {
               if (node_jobs[i]==-1)// node is available, assign it.
               {
                  node_jobs[i]=job_id;
                  task_list.all_jobs[job_id].nodes[count]=i;
                  count++;
               }
               i++;
            }
            nodes_available-=task_list.all_jobs[job_id].nodes_needed;
            // nodes assigned, create event and populate list.
            int event=0;
            double next_event_time=time+task_list.all_jobs[job_id].restart;
            event_heap.push(job_id, next_event_time, event,
task_list.all_jobs[job_id].failures);
```

```
            // done with that, move on to the next of creating the recovery item.
            double finish_time=time+ compute_needed_time(job_id);
            resources_freeing.push(finish_time, job_id,
task_list.all_jobs[job_id].failures, count, task_list.all_jobs[job_id].nodes);
            task_list.all_jobs[job_id].start_time=time;
            task_list.all_jobs[job_id].temp_value=time;
            //System.out.println("Job "+job_id+" is now running at: "+time);
        }
    void run_normal_computation(int job_id, int run, double time)
        {
            // first verify that a failure did not occur before the normal execution starts.
            if (task_list.all_jobs[job_id].failures==run)
            {
                //System.out.println("Job "+job_id+" has started computation at: "+time);
                int event=5;
                double next_event_time=time+task_list.all_jobs[job_id].interval;
```

```
                if (task_list.all_jobs[job_id].solution_time-
task_list.all_jobs[job_id].last_checkpoint<=task_list.all_jobs[job_id].interval)
                {
                   event=2;
                   next_event_time=time+task_list.all_jobs[job_id].solution_time-
task_list.all_jobs[job_id].last_checkpoint;
                } // job will not checkpoint again before completing, just finish it.
                event_heap.push(job_id, next_event_time, event,
task_list.all_jobs[job_id].failures);
                // done with that, move on to the next of creating the recovery item.
                // DONE.
             }
             // if not, we don't need to do anything.  It is already done.
          }
       double compute_start(int job)
          {
             double return_time=0;
```

```
lnode resource_frees=resources_freeing.return_first();

//int top_job=schedule_heap.heap_array[0].index;

int nodes_needed=task_list.all_jobs[job].nodes_needed-nodes_available;

while ((nodes_needed>0)&&(resource_frees!=null)) // check when the
highest priority job can run

{

   //System.out.println("job "+resource_frees.job+":
"+resource_frees.nodes[0]);

   int a_node=resource_frees.nodes[0];

   int node_job=node_jobs[a_node]; // check zero node to verify the job is
still on the same node

   int job_run=task_list.all_jobs[resource_frees.job].failures; // check
itterations to verify a failure didn't occur resulting in an outdated entry.



   if ((resource_frees.job==node_job)&&(resource_frees.run==job_run))
//validate that the job has not already terminated or failed

   {

      nodes_needed-=resource_frees.node_count;
```

```
            return_time=resource_frees.time;
            resource_frees=resources_freeing.return_next();
            //nodes_needed-=resource_frees.node_count;
        }
        else // the entry is outdated, kill it with fire!!!! I love efficiency.
        {
            resources_freeing.remove_node();
            resource_frees=resources_freeing.return_current();
        }
    }

    return return_time;
}
double compute_top_start()
{
    if (schedule_heap.last!=-1) //make sure the heap has at least one item.
    {
```

```
                double return_time=0;

                lnode resource_frees=resources_freeing.return_first();

                int top_job=schedule_heap.heap_array[0].index;

                int nodes_needed=task_list.all_jobs[top_job].nodes_needed-
nodes_available;

                while ((nodes_needed>0)&&(resource_frees!=null)) // check when the
highest priority job can run

                {

                   //System.out.println("job "+resource_frees.job+":
"+resource_frees.nodes[0]);

                   int a_node=resource_frees.nodes[0];

                   int node_job=node_jobs[a_node]; // check zero node to verify the job is
still on the same node

                   int job_run=task_list.all_jobs[resource_frees.job].failures; // check
itterations to verify a failure didn't occur resulting in an outdated entry.

                   if ((resource_frees.job==node_job)&&(resource_frees.run==job_run))
//validate that the job has not already terminated or failed

                   {
```

```
            nodes_needed-=resource_frees.node_count;
            return_time=resource_frees.time;
            resource_frees=resources_freeing.return_next();
            //nodes_needed-=resource_frees.node_count;
        }
        else // the entry is outdated, kill it with fire!!!! I love efficiency.
        {
            resources_freeing.remove_node();

            resource_frees=resources_freeing.return_current();
        }
    }
    return return_time;
}
else
{
    return -1;
```

```
        }
    }
    double compute_needed_time(int job_id)
    {
        // completion time can be estimated on assignment based on
        // estimated checkpoints= (1+e^(checkpoint latency/(checkpoint interval-checkpoint latency)))*(solution time/checkpoint interval)
        // solution time + (latency*estimated checkpoints)
        job j=task_list.all_jobs[job_id];
        double s_time=j.solution_time-j.last_checkpoint;
        //double estimated_checkpoints=(1+Math.exp(j.latency/(j.interval-j.latency)))*(s_time/j.interval);
        double estimated_checkpoints=(s_time/j.interval);
        double compute_time= s_time+j.latency*estimated_checkpoints+j.restart;
        return compute_time;
    }
    void add_job(int job_id, double now)
```

```
{
    // Adds job to schedule after checking if the job can already run in the
    // current configuration and resources.
    // No need to check other, higher priority jobs first,
    // because they would already be running if they could.
    // Only need to check when the highest priority job will be able to run.
    // Need to store when (roughly) resources will become available in a linked
list.
    // completion time can be estimated on assignment based on
    // solution time + (latency*estimated checkpoints)
    // estimated checkpoints= (1+e^(checkpoint latency/(checkpoint interval-
checkpoint latency)))*(solution time/checkpoint interval)
    // external flag in jobs or nodes will mark whether the resources have already
been freed.
    // on next resource check, resource commitments will be freed if the check
fails.
    // Linked list will have to contain node, job, and job run/failure number.
    // Increment priorities of existing jobs first:
```

```
double delta=now-last_add;
last_add=now;
schedule_heap.increment_all(-delta);
// run job if we can
// first check if jobs are waiting, if not, run now.
if (schedule_heap.last==-1) //empty, check for available resources.
{
   if (nodes_available>task_list.all_jobs[job_id].nodes_needed) // it can run
now
   {
      run_job(job_id,now);
   }
   else // can't run yet, add to scheduler if it can run on the system.
   {
      if (task_list.all_jobs[job_id].nodes_needed<=NODE_COUNT)
      {
         schedule_heap.push(job_id,0);
```

```
                    //System.out.println("Job added: "+job_id);
                }
                else // Error: the job requires more nodes than are in the system.
                {
                    System.err.println("Job "+job_id+" requires more nodes
("+task_list.all_jobs[job_id].nodes_needed+") than are in the system
("+NODE_COUNT+").");
                }
            }
        }
        else // check if job can complete before highest priority job can start running.
        {
            if (task_list.all_jobs[job_id].nodes_needed<nodes_available) // verify there
are enough idle nodes to run job
            {
                double top_runs_in= compute_top_start()-now;
                double time_needed=compute_needed_time(job_id);
```

```
                if (time_needed<top_runs_in) // job will complete before top job will be
able to start.  Run now.
                {
                   run_job(job_id,now);
                }
                else
                {
                   schedule_heap.push(job_id,0);
                }
             }
             else //can't finish in time.  Push into heap.
             {
                schedule_heap.push(job_id,0);
                System.out.println("Job added: "+job_id);
             }

          }
```

```
}

void job_finished(int id, double now, int run)

{

   // frees resources and finishes recording stuff, then checks if a job can run.

   // update execution time, free nodes then run scheduler.

   if (run==task_list.all_jobs[id].failures) // verify that it isn't an old event

   {

      System.out.println("Job "+id+" is finished at: "+now);

      task_list.all_jobs[id].execution_time+=now-
task_list.all_jobs[id].start_time;

      for (int i=0;i<task_list.all_jobs[id].nodes_needed;i++)

      {

         node_jobs[task_list.all_jobs[id].nodes[i]]=-1;

         free_resources[task_list.all_jobs[id].nodes[i]]=true;

      }

      finished++;

      nodes_available+=task_list.all_jobs[id].nodes_needed;
```

```
                //System.out.println("Nodes available: "+nodes_available);
                // job's done. No events to set up.  Just run the scheduler.
                run_scheduler(now);
            }
            // old events are ignored.  Nothing else needed.
        }
        void checkpoint_starts(int job_id, double now, int run)
        {
            if (run==task_list.all_jobs[job_id].failures)
            {
                //System.out.println("Job "+job_id+" checkpoints at: "+now);
                //System.out.println("Computation progress:
"+task_list.all_jobs[job_id].last_checkpoint);
                // store the computation time completed (now-temp) in the job data temp
value
                // the current value is the last checkpoint end time just subtract from now.
```

```
                task_list.all_jobs[job_id].temp_value=now-
task_list.all_jobs[job_id].temp_value;


task_list.all_jobs[job_id].next_checkpoint=now+task_list.all_jobs[job_id].interval;
                // next line is a placeholder, may have to change if I make the latency
variable.
                double checkpoint_ends=now+task_list.all_jobs[job_id].latency;
                //System.out.println("Job "+job_id+" checkpoint will end at:
"+checkpoint_ends);
                // push checkpoint finish event
                //
                event_heap.push(job_id, checkpoint_ends, 6, run);
                // slap a big ol' done stamp on this one.  This is all we need to do.
            }
        }
        void checkpoint_ends(int job_id, double now, int run)
        {
            if (run==task_list.all_jobs[job_id].failures)
```

```
                {

                    // increment last checkpoint by temp value in the job data

                    // store the current time in the job data temp value.

                    task_list.all_jobs[job_id].checkpoint_writes++;


task_list.all_jobs[job_id].last_checkpoint+=task_list.all_jobs[job_id].interval;

            /*          if (job_id==0) // debugging code

                    {

                        System.out.println("Job "+job_id+" ends checkpoint
"+task_list.all_jobs[job_id].checkpoint_writes+" at: "+now);

                        System.out.println("Computation progress:
"+task_list.all_jobs[job_id].last_checkpoint);

                        System.out.println("Execution time:
"+(task_list.all_jobs[job_id].execution_time+now-task_list.all_jobs[job_id].start_time));

                    }

            */

                    task_list.all_jobs[job_id].temp_value=now;

                    //double next_checkpoint=task_list.all_jobs[job_id].next_checkpoint;
```

```
                double next_checkpoint=now+task_list.all_jobs[job_id].interval;

                job cur_job=task_list.all_jobs[job_id];

                if (cur_job.solution_time-cur_job.last_checkpoint<=next_checkpoint-now)
// checking if the job will finish before next checkpoint

                {

                   // job will finish before checkpoint (unless failure occurs.

                   event_heap.push(job_id, now+cur_job.solution_time-
cur_job.last_checkpoint, 2, run);

                }

                else

                {

                   // push next checkpoint event

                   //

                   event_heap.push(job_id, next_checkpoint, 5, run);

                   // slap a big ol' done stamp on this one.  This is all we need to do.

                }

             }
```

```
}
void node_fails(int node,double now)
{
   // check if job is running on node, stops it if it is.
   // if not, it is just removed from the available resources.
   //
   if (node_jobs[node]>=0) // if a job is running on the node, free resources
   {
      // need to kill job, then update nodes the job was running on to indicate
that
      // they are available.
      // Killing a job requires incrementing failure and updating execution time,
      // updating nodes indicates updating free resources and node jobs.
      int job_index= node_jobs[node];
      task_list.all_jobs[job_index].failures++;
      task_list.all_jobs[job_index].execution_time+=now-
task_list.all_jobs[job_index].start_time;
```

```
                /*if (job_index==0)
                {
                   //System.out.println("Execution time for job "+job_index+" updated
to:"+ task_list.all_jobs[job_index].execution_time);
                }
                */
                // free resources
                for(int i=0;i<task_list.all_jobs[job_index].nodes_needed;i++)
                {
                   free_resources[task_list.all_jobs[job_index].nodes[i]]=true;
                   //System.out.println(task_list.all_jobs[job_index].nodes[i]);
                   node_jobs[task_list.all_jobs[job_index].nodes[i]]=-1;
                }
                free_resources[node]=false; // failed node is not free, guess I need this
afterall
                nodes_available+=task_list.all_jobs[job_index].nodes_needed;
                // Add job to schedule heap.
```

```
                //add_job(int job_id, double now, double delta)
                add_job(job_index,now);
                //schedule_heap.push(job_index, 0);
            }
        /*      else
            {
                System.out.println("No job running on node when failure occured. "+
node_jobs[node]);
            }
         */
            node_jobs[node]=-2; // node down flag, different from node idle flag.
            nodes_available-=1;
            // schedule repair event here <--------------------------------------
            // schedule repair event here <--------------------------------------
            // this means that the event heap will have to move to the scheduler class!
            double repair_time=now+repairs[node][(f_index[node]-1)];  // placeholder
for now, need margin adder based on LANL logs
```

```
    event_heap.push(node, repair_time, 4);
    // run scheduler here <-------------------------------------
    // run scheduler here <-------------------------------------
    run_scheduler(now);
}
void node_repaired(int node, double now)
{
    if (node_jobs[node]==-2) // verify that the node is in a failure state.
    {
        // add node to resource list
        node_jobs[node]=-1;
        nodes_available+=1;
        // schedule next failure
        double temp_time=next_failure(node,now);
        if (temp_time>0)
            event_heap.push(node, temp_time,3);
        // Run shceduler.
```

```
            run_scheduler(now);
        }
        else
        {
            System.out.println("Node "+node+" repair failed.  Node was already
active.");
        }
    }
    boolean isempty()
    {
        if (schedule_heap.last==-1)
            return true;
        else return false;
    }
    void run_scheduler(double now)
    {
        // run the scheduling algorithm and find out if it can run another program
```

```
// repeats process until no other posibilities can run
// problems: it needs to check the entire heap for jobs that can run
// This essentially means destroying the heap and rebuilding it.
// This would not be efficient if we popped then pushed into a second
// heap then simply pop and push back when done.
// Instead, I think it would be best to copy the existing heap and just
// allow this function to rip through the heap copy.
// Since the jobs are ordered by priority, it only takes one pass to try
// to fill all resources.  If something didn't fit on the first pass,
// it won't fit on the second.
// copy method won't work as the jobs will stay in the scheduler
// we need to fix that by using the pull and push method into a second heap.
// slightly less efficeint, but it will work.
priority_queue q_unrun=new priority_queue(schedule_heap.size);
// first check if head can run, if not, check *WHEN* head can run.
// repeat until head can't run, queue is empty, or all resources full.
pq_node top=schedule_heap.pop();
```

```
            double r_time;
            while
((top!=null)&&(task_list.all_jobs[top.index].nodes_needed<=nodes_available)) // top job
can run
            {
               //System.out.println("a top: "+top.index);
               run_job(top.index,now);
               top=schedule_heap.pop();
            }
            if (top!=null) // the heap still has jobs in it
            {
               // now check if lower priority programs can run on available resources
               // before the head will run.  Keep going until queue is empty.
               //System.out.println("top: "+top.index);
               r_time=compute_start(top.index); // find the start time of the top
               q_unrun.push(top.index,top.priority);
               top=schedule_heap.pop();
```

```
                while (top!=null)
                {
                   int job_id=top.index;
                   job cur_job=task_list.all_jobs[job_id];
                   // check if backfilling can allow jobs to run before top job.
                   if
((cur_job.nodes_needed<=nodes_available)&&(compute_needed_time(job_id)<r_time))
                   {
                      //System.out.println("b top: "+top.index);
                      run_job(job_id,now); // backfill
                   }
                   else
                   {
                      //System.out.println("c top: "+top.index);
                      q_unrun.push(top.index,top.priority); // doesn't meet criterion, put in
queue.
                   }
```

```
            top=schedule_heap.pop();
            //job_id=top.index;
            //cur_job=task_list.all_jobs[job_id];
          }
          // schedule heap is now empty, replace with q_unrun
          schedule_heap=q_unrun.make_copy();
        }
    }
}
public class Main {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args)
    {
        // TODO code application logic here
        int priority;
```

```
int value;



//System.out.println("test1");

scheduler s=new scheduler(parameters.start_time);

//jobs tasks=new jobs(); //load and populate the jobs.

//Random generator=new Random();

//priority_queue my_queue=new priority_queue();

pq_node top;

//priority_queue event_heap=new priority_queue();

//System.out.println("test1a");

// code below is just a test of the queue to verify that it works.

for (int i=0;i<s.task_list.size;i++)

{

   // for each job, schedule the arrival

   //System.out.println(s.task_list.size+" Job "+ i+":
"+s.task_list.all_jobs.toString());

   s.event_heap.push(i, s.task_list.all_jobs[i].arrival_time, 1);
```

```
            //priority=generator.nextInt(1000);
            //value=generator.nextInt(1000)+1000;
            //my_queue.push(value,priority);
        }
        int breaker=0;
        //System.out.println("test1b");
        top=s.event_heap.pop();
        //priority_queue scheduler_heap=new priority_queue();
        double last_time=0;
        double last_finish=0;
        //System.out.println("test1c");

        while((top!=null)&&(s.finished<s.task_list.size))
        {
            //System.out.println(top.priority+":"+top.index);
//  /*
```

```
switch (top.event)
{
   case 0:
      // need to set up system to schedule checkpoint or completion
      // of the program at the time specified by start-up that the
      // scheduler initiates
      // Simple way:
      // run job algorithm assigns the nodes and sets start time
      // as it already does, but instead of scheduling a checkpoint or
      // completion, it schedules program start which then sets up the
      // checkpoint or completion event.  In other words,
      // just split it in two.
      s.run_normal_computation(top.index, top.run, top.priority);
      break;
   case 1: // job queued for scheduler
      System.out.println("Job "+top.index+" is scheduled at:
"+top.priority);
```

```
                    // Update priorities for jobs already in queue
                    // We want jobs in the queue longer to have a higher priority,
                    // but we are using a min heap, so we use negative values.
                    double delta=last_time-top.priority; // temporary for now.
                    // add job to the shceduler
                    s.add_job(top.index, top.priority);
                    last_time=top.priority;
                    break;
                case 2: // job finishes
                    // update available resource list.
                    // pull job related events from queue (or flag job as finished so next
event is ignored, this may be more efficient).
                    // update final execution time.
                    // run scheduler to see if enough resources have freed to run another
task.
                    s.job_finished(top.index,top.priority,top.run);
                    last_finish=top.priority;
```

```
                    break;

                case 3: //Node fails

                    System.out.println("Node "+top.index+" fails at: "+top.priority);

                    // update total execution time of job (if any)

                    // pull job related events from queue (or flag with run number so that we can ignore them if they don't match)

                    // If job was running, add nodes that the job was running on to the available resource list.

                    // If job was running on node, run scheduler to see if another job can run on the newly available resources.

                    // Remove node from available resource list.

                    s.node_fails(top.index, top.priority);

                    break;

                case 4: //Node is restored

                    System.out.println("Node "+top.index+" is restored at: "+top.priority);

                    // Add node to available resource list.

                    // run scheduler to see if job can run on available resource list.
```

```
                    //
                    s.node_repaired(top.index, top.priority);
                    break;
                case 5: //job checkpoints
                    //System.out.println("Job "+top.index+" checkpoints at:
"+top.priority);
                    // Increment job completion event by checkpoint latency.
                    // create job finishes checkpoint event.
                    //
                    s.checkpoint_starts(top.index, top.priority, top.run);
                    break;
                case 6: //
                    //System.out.println("Job "+top.index+" finishes checkpoint at:
"+top.priority);
                    // Update last checkpoint time for job
                    // Schedule job checkpoint event (current_time+interval-latency).
                    //
```

```
                    s.checkpoint_ends(top.index, top.priority, top.run);

                    break;

                }

            // */

                breaker++;

                top=s.event_heap.pop();

            }

            System.out.println();

            for (int j=0;j<s.task_list.size;j++)

            {

                System.out.println("Job: "+j);

                System.out.println("Solution time: "+s.task_list.all_jobs[j].solution_time);

                System.out.println("Execution time:
"+s.task_list.all_jobs[j].execution_time);

                System.out.println("Failures: "+s.task_list.all_jobs[j].failures);

                System.out.println("Checkpoints:
"+s.task_list.all_jobs[j].checkpoint_writes);
```

```
                System.out.println();
                //System.out.println("Failures: "+s.task_list.all_jobs[j].+"\n");
            }
            try
            {
                FileOutputStream fstream = new
FileOutputStream(parameters.output_file);
                // Get the object of DataInputStream
                PrintStream out = new PrintStream(fstream);
                out.println("job,solution_time,execution_time,failures,checkpoints");
                for (int j=0;j<s.task_list.size;j++)
                {

out.println(j+","+s.task_list.all_jobs[j].solution_time+","+s.task_list.all_jobs[j].execution
_time+","+s.task_list.all_jobs[j].failures+","+s.task_list.all_jobs[j].checkpoint_writes);
                    //out.println("Job: "+j);
                    //out.println("Solution time: "+s.task_list.all_jobs[j].solution_time);
```

```
            //out.println("Execution time: "+s.task_list.all_jobs[j].execution_time);
            //out.println("Failures: "+s.task_list.all_jobs[j].failures);
            //out.println("Checkpoints: "+s.task_list.all_jobs[j].checkpoint_writes);
            //out.println();
            //System.out.println("Failures: "+s.task_list.all_jobs[j].+"\n");
        }
        out.println("\n"+last_finish);
        out.close();
        fstream.close();
    }
    catch(Exception e)
    {//Catch exception if any
        System.err.println("Error: " + e.getMessage());
    }
    System.out.println(breaker);
  }
}
```

D. Additional Tables

Table A. 1 Logliklihood Values for LANL Failure Data K-means with Density Based Clustering.

k-value	system 2	system 16	system 18	system 19	system 20
10	-11.75742	-10.17494	-8.5393	-8.51907	-13.09569
20	-11.67481	-10.01109	-8.39141	-8.33097	-12.96425
30	-11.64115	-10.06099	-8.36368	-8.27566	-12.89122
40	-11.60658	-10.12883	-8.31638	-8.23955	-12.80634
50	-11.58049	-10.10106	-8.28931	-8.22876	-12.75247
60	-11.55606	-10.05154	-8.26154	-8.20315	-12.70995
70	-11.55206	-10.12189	-8.26325	-8.228	-12.66187
80	-11.5318	-10.14495	-8.2903	-8.21858	-12.65116
90	-11.52169	-10.20823	-8.30311	-8.27314	-12.59552
100	-11.51032	-10.20991	-8.252	-8.22331	-12.55912
110	-11.5082	-10.19877	-8.27857	-8.25398	-12.52021
120	-11.47895	-10.20076	-8.23316	-8.26087	-12.47367
130	-11.4614	-10.19815	-8.2273	-8.23104	-12.44532
140	-11.4524	-10.19961	-8.24134	-8.20077	-12.42183
150	-11.45542	-10.19655	-8.28727	-8.2559	-12.45431
160	-11.47567	-10.18669	-8.28669	-8.25881	-12.45733
170	-11.49746	-10.17018	-8.24581	-8.2545	-12.43708
180	-11.48779	-10.17274	-8.28559	-8.25324	-12.41406
190	-11.47979	-10.17174	-8.27022	-8.24828	-12.40517
200	-11.48129	-10.17103	-8.24418	-8.25712	-12.40911

Curriculum Vita

Michael Harney is the fourth child of William and Carolyn Harney. He was Born in El Paso, Texas. He got his Bachelor's degree from the University of Texas at El Paso in Biology in Dec. 1999. He returned to UTEP Twice since: once to obtain his teaching certification; and again to obtain his Master's Degree in Computer Science which he will complete in May of 2013. While working towards his Master's, he worked for UTEP's HIgh PERformance SYStems (HiPerSys) group doing research on Checkpoint I/O measurement and optimization.

Permanent address: 8522 Chinchilla Ln.

El Paso, Tx. 79907