ACM/IEEE SuperComputing Conference(SC'07),2007

Performance Under Failures of High-End Computing

Ming Wu[†], Xian-He Sun[†‡] and Hui Jin[†]
Department of Computer Science[†]
Illinois Institute of Technology
Chicago, Illinois 60616, USA
Fermi National Accelerator Laboratory[‡]
Batavia, Illinois 60510-0500
{wuming, sun, hjin6}@iit.edu

ABSTRACT
Modern high-end computers are unprecedentedly complex. Occurrence of faults is an inevitable fact in solving large-scale applications on future Petaflop machines. Many methods have been proposed in recent years to mask faults. These methods, however, impose various performance and production costs. A better understanding of faults' influence on application performance is necessary to use existing fault tolerant methods wisely. In this study, we first introduce some practical and effective performance models to predict the application completion time under system failures. These models separate the influence of failure rate, failure repair, checkpointing period, checkpointing cost, and parallel task allocation on parallel and sequential execution times. To benefit the end users of a given computing platform, we then develop effective fault-aware task scheduling algorithms to optimize application performance under system failures. Finally, extensive simulations and experiments are conducted to evaluate our prediction models and scheduling strategies with actual failure trace.

Categories and Subject Descriptors
C.4 [**Performance of Systems**]: Fault tolerance, Performance attributes, Reliability

General Terms
Performance, Reliability.

Keywords
Fault-tolerance, Failure Modeling, Application Performance.

1. Introduction
Computing capacity is essential to understand the universe and to improve the quality of life. Teraflop computers, which are capable of executing one trillion (10^{12}) floating point operations per second (TFlops), are in workplaces. Petaflop systems (10^{15}) are also on the horizon. These high-end computers are unprecedentedly complex, highly parallel with tens of thousands of CPUs, tera- or peta-bytes of main memory, and tens of peta-bytes of storage. In addition, they are often used coordinately and collectively over the Internet to support modern computing paradigms, such as Grid computing, peer-to-peer computing, and service oriented computing. While these high-end parallel and distributed systems are able to deliver unprecedented computation power, any failure of any of their components may lead to a system failure. Recent research shows that in a parallel system composed of thousands of nodes, the system failure happens several times a day [15]. As the size of parallel and distributed systems continually grows, the Mean Time to Failure (MTTF) can drop to a few hours, even minutes [9]. The impact of system failure is becoming an increasingly important factor of application performance, especially for large-scale scientific applications.

Many methods exist to increase the reliability and mitigate the performance loss of computing systems. From the hardware point of view, we may deploy replicas or adapt more expensive but more reliable hardware to increase the system reliability. From software point of view, we can provide the support of checkpointing, process migration, and program restart to mitigate the performance loss [10, 14]. Checkpointing stores a snapshot of the current application state and use it for restarting the execution in case of failure. Process migration moves a process from one node to another node. Program restart is to restart an application at another machine. They can be used collaboratively or individually to provide a fault-tolerant environment. These fault-tolerant methods mitigate the performance loss of system failures and, in the meantime, they also impose various performance and production costs. The overhead and effectiveness of the underlying fault tolerant environment determine the impact of system failures on application performance. Designing an optimal fault tolerant environment, however, is elusive and application dependent. For example, a computing node can be used in parallel processing to increase computing power, or used as a replica to increase the reliability, or reserved and kept idle for possible process migration. How to utilize this node for best performance is not a trivial question. A better understanding of faults' influence on application performance is a necessity for designing an optimal fault tolerate environment.

We conduct research in modeling and optimization of performance under failure in this study. We first develop some practical and effective performance models to predict the application completion time under system failures. These models separate the influence of failure arrival, failure repair, checkpointing period, checkpointing cost, and parallel task allocation on parallel and sequential execution times. Then, performance optimization is conducted at both the system-level

and user-level. At the system-level, we use the newly derived models to investigate the effects of different system parameters on application performance, use them as a guideline to build cost-effective fault-tolerant computing systems. At the user-level, we study how to optimize application performance under a given computing environment. Fault-aware task scheduling algorithms are developed to allocate tasks to appropriate resources. Analytical and experimental results show these modeling and optimization mechanisms work well. They provide a feasible solution for improving performance under failure of high-end computing.

This paper is organized as follows. In Section 2, we introduce a general failure model and extend it for a specific failure handling scenario where the checkpointing cost is further divided into rollback cost and storage cost. We derive formulas to calculate the optimal checkpoint period. In Section 3, we investigate the effect of different system parameters on application performance. We propose some fault-aware task scheduling algorithms in Section 4. In Section 5, simulations and experiments are conducted to verify the correctness of our analytical models and test the efficiency of the proposed scheduling algorithms. Related works are discussed in Section 6. Finally, we conclude this study and discuss future works in Section 7.

2. The Models of Performance under Failure

In general, a system failure is a non-deterministic event and can not be described by deterministic models. Probability models are thus applied in this study. We use an M/G/1 process [7] to describe system failures. Based on the common assumption in reliability engineering that the time for different components in a computer system to fail is usually exponentially distributed (or the occurrence of failures is essentially random) [4, 19], we assume that the arrival of failures follows a Poisson distribution with λ_f, and the downtime of failures follows a general distribution with mean μ_f and standard deviation σ_f, which is a generalization on the exponential downtime distribution used in [4,19]. Notice that λ_f is the reverse of MTBF (Mean Time Between Failure) and μ_f reflects MTTR (Mean Time To Recovery). Checkpointing has been widely used as the key fault tolerant technology in high-end computing. We focus on modeling the performance influence of checkpointing in this study. Depending on the implementation of checkpointing mechanisms, checkpointing can be performed at either system-level or application-level. The checkpointing frequency can be fixed (periodically) or adaptive (adjust at run time). This will lead to different checkpointing cost. To address this concern, we first build a general failure model where the checkpoint/recovery cost follows a general distribution. Then, we assume checkpointing is performed periodically and model the impact of different checkpointing parameters on the application performance separately. This more detailed model can be used to determine the best checkpointing strategy. Its derivation process also illustrates how to build other specific failure models from the general model.

Let W denote the application workload in terms of execution time. We assume that the checkpoint/recover cost follows a general distribution with mean μ_c and standard deviation σ_c. Here the checkpoint/recover cost refers to the time required by the system to recover the application from the last checkpoint to the failure point. In this situation, suppose the application's execution is interrupted by failures S times, the completion time of the application can be expressed as

$$T = X_1 + Y_1 + Z_1 + X_2 + Y_2 + Z_2 + ... + X_s + Y_s + Z_s + L \quad (1)$$

where $X_i (1 \le i \le S)$ are the computing time consumed by the application, $Y_i (1 \le i \le S)$ are the downtime of system failures, $Z_i (1 \le i \le S)$ are the checkpoint/recover cost after failure interruption and L is the execution time of the last application process that finishes the application. Since we have $w = X_1 + X_2 + ... + L$, we get

$$T = w + Y_1 + Y_2 + ... + Y_S + Z_1 + Z_2 + ... + Z_s \quad (2)$$

Under the assumption of the M/G/1 failure processing, we can get the first and second moments of $Y_i (1 \le i \le S)$ using existing results from queueing theory,

$$E(Y_i) = \frac{\mu_f}{1 - \lambda_f \mu_f}, \qquad E(Y_i^2) = \frac{\sigma_f^2 + \mu_f^2}{(1 - \lambda_f \mu_f)^3}.$$

The mean and variance of T can be obtained through the following expression (See Appendix for proof):

$$E(T) = (\frac{1}{1 - \lambda_f \mu_f} + \lambda_f \mu_c) w \quad (3)$$

$$V(T) = (\frac{\mu_f^2 + \sigma_f^2}{(1 - \lambda_f \mu_f)^3} + \mu_c^2 + \sigma_c^2 + 2\frac{\mu_f \mu_c}{1 - \lambda_f \mu_f}) \lambda_f w \quad (4)$$

.

The cumulative distribution function of the application completion time is expressed as:

$$\Pr(T \le t) = \Pr(T \le t \mid S = 0) \Pr(S = 0) + \Pr(T \le t \mid S > 0) \Pr(S > 0)$$

.Since $\Pr(S = 0) = e^{-\lambda_f w}$, we have

$$\Pr(T \le t) = \begin{cases} e^{-\lambda_f w} + (1 - e^{-\lambda_f w}) \Pr(U(S) \le t - w \mid S > 0) & \text{if } t \ge w \\ 0, & \text{otherwise} \end{cases} \quad (5)$$

where

$$U(S) = \begin{cases} 0, & \text{if } S = 0 \\ Y_1 + Z_1 + Y_2 + Z_2 ... + Y_S + Z_S, & \text{if } S > 0 \end{cases}$$

,

and (See Appendix for proof)

$$E(U(S) \mid S>0) = \frac{\lambda_f(\mu_f+\mu_c) - \lambda_f^2 \mu_c \mu_f}{(1-\lambda_f \mu_f)1-e^{-\lambda_f w}} w \quad (6)$$

$$V(U(S) \mid S>0) = (\frac{\mu_f^2+\sigma_f^2}{(1-\lambda_f\mu_f)^3} + \mu_c^2 + \sigma_c^2 + 2\frac{\mu_f\mu_c}{1-\lambda_f\mu_f}) \frac{\lambda_f w}{1-e^{-\lambda_f w}} \quad (7)$$

The first term on the right-side of Formula (5) is the performance without failure interruption. The second term, $(1-e^{-\lambda_f w})\Pr(U(S) \le t-w \mid S>0)$, is the performance with the failure interruptions. Based on [6], the Gamma distribution is an appropriate distribution to describe the $\Pr(U(S) \le u \mid S>0)$.

When the application is a parallel task, its execution time, T, is the maximum of its sub-task completion times. In this study, we assume that the parallel job is composed of one single parallel phase with no communication or synchronization requirements other than the final synchronization, which occurs when all of the tasks have completed. Assuming *m* nodes are used for parallel computing, T can be expressed as

$$T = Max\{T_k, k = 1,2,...m\}.$$

Assuming the failure occurrences of different nodes are independent, the probability of the parallel task finish with time t is equal to

$$\Pr(T \le t) = \Pr(Max\{T_k, k=1,2,...m\} \le t) = \prod_{k=1}^{m} \Pr(T_k \le t) \quad (8)$$

where T_k is the subtask execution time on node m_k.

Checkpointing is often conducted periodically with some fixed interval. A challenging problem, then, is how to determine an appropriate checkpoint period. The selection of an optimal checkpoint period may depend on many factors, such as the time to save and restore the application status, the arrival rate of failures, the failure downtime distribution, and the users' performance requirement. To identify the impact of these factors on checkpointing period, we need to further explore our proposed models. In formula (1), the checkpointing cost refers to all the system overhead incurred in order to recover application status from the last check point. It may include the time needed for the system to perform checkpointing and the time needed for the application to run from the last checkpoint to the failure point. To distinguish these two types of cost, we name the former as storage cost and the later as rollback cost. Thus, the application completion time under failure can be expressed as

$$T = X_1 + Y_1 + Z_1 + X_2 + Y_2 + Z_2 + ... + X_s + Y_s + Z_s + L + \sum_{i=1}^{J} T_c(i) \quad (9)$$

where J is the total number of checkpoints during application execution and $T_c(i)$ is the time needed for the system to perform checkpoint at i^{th} time. The storage cost consists of the cost of store and reload of the computing image. We can assume that the storage cost is a constant independent of the length of the checkpointing period. Suppose that the checkpointing period is γ and the storage cost is α, we get

$$T = w + \frac{w}{\gamma}\alpha + Y_1 + Y_2 + ... + Y_S + Z_1 + Z_2 + ... + Z_s \quad (10)$$

Because the failure arrival follows Poisson distribution, the time between the last checkpoint and the first failure point (rollback cost) during a checkpointing period follows exponential distribution. Thus we can calculate the mean of rollback cost,

$$\mu_c = \frac{1-e^{-\lambda_f\gamma} - \lambda\gamma e^{-\lambda_f\gamma}}{\lambda_f(1-e^{-\lambda_f\gamma})} \quad (11)$$

Then, we have

$$E(T) = (\frac{1}{1-\lambda_f\mu_f} + \frac{\alpha}{\gamma} + \frac{1-e^{-\lambda_f\gamma} - \lambda\gamma e^{-\lambda_f\gamma}}{(1-e^{-\lambda_f\gamma})}) w \quad (12)$$

$$V(T) = (\frac{\mu_f^2+\sigma_f^2}{(1-\lambda_f\mu_f)^3} + \mu_c^2 + \sigma_c^2 + 2\frac{\mu_f\mu_c}{1-\lambda_f\mu_f})\lambda_f w \quad (13)$$

For given λ_f, μ_f, α, and w, $E(T)$ can be viewed as a function of γ. By solving the function $\frac{\partial E(T)}{\partial \gamma} = 0$, we can find the optimal checkpointing period to minimize the expected application execution time.

3. Effect of System Parameters on Parallel Task Completion Time

Equipped with the models derived in Section 2, we are now ready to examine the effects of different system parameters on application performance.

Using the probability expression (5) in Section 2, we are able to evaluate the distribution of parallel task completion time T analytically. The impact of system parameters on the mean and STD (standard deviation) of application completion time is investigated. Recall, the Gamma distribution can be used to approximate the random variable $U_k(S) \mid S_k > 0$ [6], and the mean and standard deviation is given by expressions (12) and (13). Six parameters are examined in our computations: failure rate, failure downtime, number of nodes, storage cost, workload on each node, and checkpointing period. In each figure, two of the six parameters are examined. The remaining parameters, if not specified, are assumed as the default values. The default values are listed below:

- Failure rate on each node =1/(512) hour
- Failure down time=2 hours
- Number of Nodes = 128
- Storage cost α = 0.2 hour.
- Workload on each node = 128 hours
- Checkpointing period = 2 hours

The default values for failure arrive rate and failure downtime are obtained by studying the failure data provided by Los Alamos National Lab [8]. The failure data is collected from 22 high-

performance computing systems between 1996 and November 2005. We set the default value of the storage cost based on the actual performance of BlueGene/L [12].

Figure 1 illustrates the effect of coefficient of failure rate and downtime on mean parallel completion time. The application completion time is reduced with the decreasing of either failure rate or downtime. When the failure rate is high, reducing failure rate can lead to a significant improvement on application performance. However, if the failure rate is low, the effect of failure rate on application completion time is negligible. A similar conclusion can be found for the failure downtime. From Figure 1, we can also observe that failure downtime has more effect on application performance than failure rate. For instance, at the point where the failure rate is 1/256 per hour and the down time is 8-hour, if we reduce the downtime to 4-hour, the application expected completion time decrease from 326.0 hour to 225.5 hour. If we reduce the job failure rate to 1/512, the application completion is only reduced to 266.1 hour.

The effect of storage cost and failure rate on mean parallel completion time is shown in Figure 2. The parallel task completion time decreases when the storage cost decreases for a given failure rate. For a given storage cost, reducing the failure rate can lead to the decrease of task completion time. However, when the storage cost increases, it becomes a dominant factor in determining the application performance. There is a near linear relation between parallel completion time and storage cost when the failure rate is very low for a given workload. This indicates when failure rate is lower than a threshold, its effect becomes negligible, and more attention should be paid to reducing the storage cost.

The parallel task completion time for different checkpointing periods is given in Figure 3. We can find that there is an optimal point to achieving the best task completion time for each failure rate. Similar result has been observed for failure downtime and storage cost. Please note that the value of this optimal checkpointing period is independent of workload and the number of nodes. Figure 4 plots the parallel completion time with different number of nodes and failure rate with the fixed default checkpointing period. We can see for most situations, the number of nodes has more effect on the application performance than failure rate. For instance, at the point where the failure rate is 1/256 per hour and the system size is 256-nodes, if we increase the system size to 512-nodes, the application expected completion time decrease from 42.5 hour to 26.4 hour. If we reduce the job failure rate to 1/512, the application completion is only reduced to 40.1 hour. Only when the system size is very large, the failure rate per node may play an important role in impacting the parallel task completion time.

The effect of the number of nodes and the application workload on the STD of parallel task completion time is given in Figure 5. We notice that, when the number of nodes and the workload at each node are both kept at a higher level, the changes of the STD of parallel task completion time is minor. This observation indicates that our models remain stable for large-scale computing systems.

Figures 1-5 illustrate the influence of failures on application performance. Using the prediction formulas of performance under failure, we can design appropriate fault tolerant environments and are able to answer the question posted in Section 1: given the possibility to be used in computing, replica, and migration, how should we utilize a computing node for best performance under failure. Let us try to solve it. Suppose the system is homogeneous. The system size is N and the total application workload is W. If the node is used for computing, according to formula (8), the expected application execution time is $\Pr(T \le t) = \prod_{i=1}^{N+1} \Pr(T_k \le t)$ where the subtask workload is reduced from W/N to $W/(N+1)$. If the node is used as a replica, then the expected application time is $\Pr(T \le t) = \prod_{i=1}^{N-1} \Pr(T_k \le t) * (1 - \Pr(T_N > t) \Pr(T_{N+1} > t))$, where the subtask workload remains as W/N. If the node is reserved as a backup resource, the expected application execution is $\Pr(T \le t) = \prod_{i=1}^{N} \Pr(T_k \le t)$, where the failure repair time is reduced from μ_f to $(1-P_1)\mu_f$. Here P_1 is the probability that only one node in the system is failed at any time. Notice that if there is only one node failed, we can always migrate the application from the failed node to the backup node without waiting for the repair of failure. By calculating the application completion time with different fault handling strategies, we can find the most appropriate method for best application performance. For example, when N =10, W =1280 hours, and the repair time is 0.5 hour, the expected application execution time is 128.60, 141.29 and 140.83 hours, if the extra node is used for computing, replica and backup, respectively. In this case, we can gain a better performance by adding the node as a computing node. However, if N =100, W =6400 hour, and the repair time is 4 hours, the execution time for the three approaches is 108.22, 109.01, and 97.05 hours. We find that using the node as backup resource can achieve the best performance.

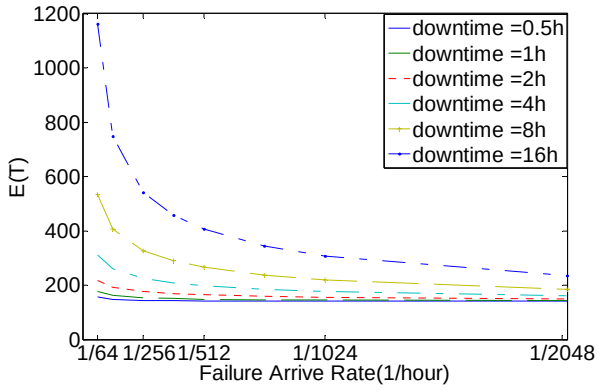

Figure 1. Mean of the application completion time with different failure rates and downtimes

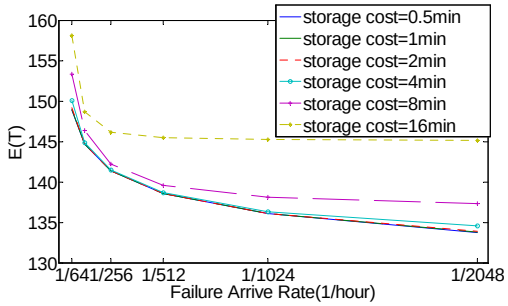

Figure 2. Mean of the application completion time with different failure rates and storage costs

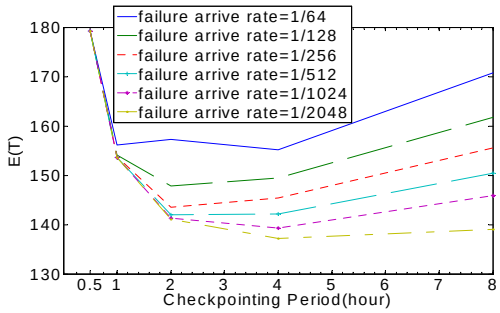

Figure 3. Mean of the application completion time with different checkpointing periods and failure rates

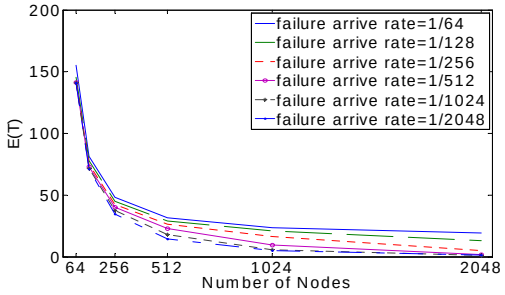

Figure 4. Mean of the application completion time with different number of nodes and failure rates

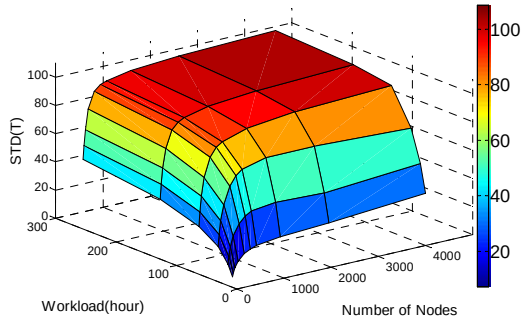

Figure 5. STD of the application completion time with different workload and system sizes

4. Fault-aware Task Partition and Scheduling

By quantitatively analyzing the effect of different system reliability parameters on application performance, the proposed models can provide a guideline to choose the best system configuration and failure handling mechanisms. However, most of the users may not have the authority to configure a system. Even if they do, different computing resources may have different failure rates and fault tolerance costs. Scheduling tasks appropriately among computing resources will reduce execution time. For this reason, we study fault-aware task scheduling in this section.

In a parallel and distributed environment, task scheduling is responsible for selecting a set of appropriate machines and distributing the application workload onto each resource. A good scheduling decision can significantly improve performance. Unfortunately, most existing task scheduling systems don't consider the impact of failure on application performance [1]. In conventional scheduling algorithms, a tacit assumption is that the underlying computing system is reliable and fault penalty is not a factor of performance. When fault penalty is a factor of performance, as for large-scale parallel and distributed systems, conventional scheduling becomes obsolete and fault-aware scheduling algorithms need to be developed. These fault-aware scheduling algorithms can be derived based on the failure models introduced in Section 2.

Several fault-aware scheduling algorithms have been developed to serve different needs. Fault-aware scheduling of a sequential job is relatively simple: estimate the expectation and variance of the application completion time on all machines and then choose the machine with the smallest sum of these two. In parallel system, the parallel task completion time is decided by the maximum subtask completion time. We use the equal-time partition strategy [18], which partitions the parallel task w into subtasks with workload w_k for machine m_k such that the same mean subtask completion time can be reached at each machine. The subtask workload w_k is calculated as

$$w_k = \frac{(\frac{1-\rho_{f,k}}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}})\tau_k}{\sum_{k=1}^{q}\tau_k(\frac{1-\rho_{f,k}}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}})}w \qquad (14)$$

where $\rho_{f,k}=\lambda_{f,k}\mu_{f,k}$ and $\rho_{c,k}=\lambda_{f,k}\mu_{c,k}$ (See Appendix for proof). $\mu_{f,k}$ and $\mu_{c,k}$ are the mean downtime and checkpoint/recover time on machine m_k.

Figure 6 shows the scheduling algorithm for a single parallel application with a given degree of parallelism (under our assumption of one subtask for one machine, the degree of parallelism equals the number of machines or subtasks used for the parallel processing). Let q denote the number of available machines and p denote the number of subtasks. We need to go through C_q^p possible machine sets to get an optimal task scheduling decision. When the application workload can be

partitioned arbitrarily into any number of subtasks, we can develop a scheduling algorithm for optimal parallel processing. To achieve an optimal scheduling plan, we need to search 2^q possible degree of parallelisms and machine combinations. The cost is quite high when the machine set is large.

Assumption: solving a parallel task on *p* machines. Each machine only hosts one subtask.
Objective: Scheduling a parallel task on a given number of machines

---

Begin
Given a set of idle machines;
$M = \{m_1, m_{2,} \ldots m_q\}$
List all the possible sets of machines, $S = \{S_1, S_2, \ldots S_z\}$, $S_i \subset M$ and $|S_i| = p$;
$p' = 1$;
For each machine set S_k $(1 \le k \le z)$,
Use equal-time partition to assign the application workload to each machine in S_k;
Calculate the mean and coefficient of variation of the remote task completion time, $E(T_{S_k})(1 + Coe.(T_{S_k}))$, approximating $\Pr(U(S_k) \le u \mid S_k > 0)$ with the Gamma distribution;
If $E(T_{S_{p'}})(1 + Coe.(T_{S_{p'}})) > E(T_{S_k})(1 + Coe.(T_{S_k}))$, then $p' = k$;
End For
Assign the parallel task to the machine set $S_{p'}$;

End

Figure 6. Parallel fault-aware task scheduling algorithm with a given number of subtasks

A heuristic task-scheduling algorithm, as shown in Figure 7, is proposed to find a near optimal solution with a reasonable cost. The algorithm has two basic steps. The first step is to sort each powerful machine according to $\frac{(1-\rho_{f,k})\tau_k}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}}$. $\frac{(1-\rho_{f,k})}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}}$ indicates how much percent of a machine's CPU resource is available for the application and τ_k is the machine's computing power. So the production of $\frac{(1-\rho_{f,k})}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}}$ and τ_k stands for the amount of the machine's computing power available to the application execution. A higher value $\frac{(1-\rho_{f,k})\tau_k}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}}$ of a machine indicates the corresponding machine has more available computing power and thus should be considered first. The second step of this algorithm is to use the bi-section search to find the local optimal based on the ordering.

Assumption: a parallel task can be partitioned into any size of subtasks. Each subtask will be assigned to a machine respectively.
Objective: scheduling a parallel task heuristically to reach a semi-optimal performance

---

Begin
List a set of idle machines in the order of their reliability over an observed time period, $M = \{m_1, m_{2,} \ldots m_q\}$;
Sort the list of idle machines in an decreasing order with $\frac{(1-\rho_{c,k})\tau_k}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}}$, $M' = \{c_1, c_{2,} \ldots c_q\}$;
$a = 1$, $b = \min\{|M'|, \frac{w}{4*(\mu_{f,k}+\mu_{c,k})}\}$;
Repeat
$c = \lfloor (a+b)/2 \rfloor$
/* $f(x)$ denotes $E(T_{C(x)})(1 + Coe.(T_{C(x)}))$ where $C(x) = \{c_1, c_{2,} \ldots c_x\}$ */
If $f(a) = \min\{f(a), f(b), f(c)\}$ **then** $b = c$
Else If $f(b) = \min\{f(a), f(b), f(c)\}$ **then** $a = c$
Else If $f(c) < f(c+1)$ **then** $b = c$
Else $a = c$
Until $a + 1 = b$
If $f(a) < f(b)$ **then**
Assign parallel task to the machine set $C(a)$;
Else Assign parallel task to the machine set $C(b)$;
End

Figure 7. A heuristic fault-aware task scheduling algorithm

5. Experimental Results

To verify the correctness and effectiveness of the newly proposed models and scheduling algorithms, we have conducted extensive experiments and simulations. We first build a simulation environment to verity the proposed failure models. We then measure their prediction accuracy with actual system failure data collected at the Los Alamos National Lab [8]. At the end, we evaluate the efficiency of the proposed fault-aware task partition and scheduling strategies.

Two performance metrics are generally used in the literature to evaluate the accuracy of a prediction model. One is percentage prediction error, which is defined as $|\frac{\Pr ediction - Measurement}{Measurement}|$ [17]. Another is square prediction error, which is defined as $(\Pr ediction - Measurement)^2$ [DiHa00, Wols98]. For large-scale applications, the square prediction error could be very big even for excellent prediction. This is especially true for scalable computing, where the square prediction error may increase with the problem size. Percentage prediction error is a more appropriate metric for large-scale applications and for scalable computing. In our experiments, we are mainly concerned about the effectiveness of failure models for large applications with different sizes. Thus we choose the percentage prediction error and simply call it prediction error throughout the study.

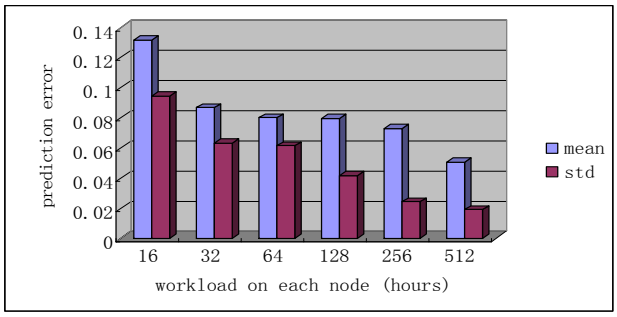

Figure 8. Mean and STD of prediction error with different workloads

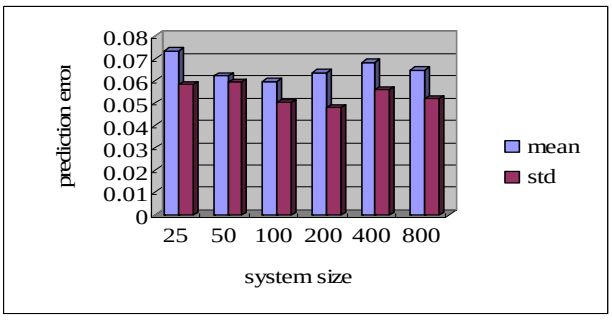

Figure 9. Mean and STD of prediction error with different machine numbers

In our simulation, the mean of failure rate on each machine ranges from $1.5*10^{-7}$ to $4.5*10^{-7}$ per second and the mean of failure downtime ranges from 2 hours to 4 hours. These values are set based on the observation of the failure rate and repair time collected in [8]. The parameters on each machine are randomly generated within their corresponding ranges so that the failure rate and downtime on each machine are different. Figure 8 plots the mean and the standard deviation of prediction error of parallel task completion time under failure. The application workload on each node ranges from 16 hours to 512 hours (without failure) and the system size is 100 nodes. We observe that the prediction error is relatively small, less than 10% for most of the situations. As the application workload increases, both mean and variation of the prediction error decrease, indicating our prediction models work well for large-scale applications. When the number of processors, or the system ensemble size, increases with problem size, the results are even more encouraging. Figure 9 gives the mean and the standard deviation of prediction error with different system size where machine number increases from 25 to 800, where the application workload increases with the system size, maintaining an average of 32-hour workload on each node. As we can see, the prediction error is all less than 10% for all system sizes. This shows the potential of the prediction of performance under failure for the upcoming Petaflops systems. In our simulation, we examine different failure downtime distributions (Lognormal, Exponential and Gamma). Similar results are observed. These results demonstrate the correctness of the analysis of failure models.

An interesting question is whether our proposed models can be applicable in real situations since it is derived based on the assumption that failure arrival is a Poisson distribution. To answer this question, we conduct experiments on the failure trace collected at the Los Alamos National Lab [8]. The LANL failure trace is collected from 22 high-performance computing systems between 1996 and November 2005. Most of these systems are large clusters of NUMA and SMP nodes [15]. Using different system parameters, i.e. the number of nodes, workload at each node, etc, we did hundreds of experiments with the LANL failure trace as the system failure and measured the average prediction error of the performance under failure. The number of nodes used in our experiments is ranged from 32 to 512. Most of the failure trace we used belongs to different LANL systems, so we can assume the failures are independent. The values for other system parameters, i.e. storage cost, checkpointing period, etc, are set the same as those in Section 3. In the experiment, we use failure data between year 2004 and 2005. According to [15], failure might have different patterns at different stages of life. For the data at [8], most systems were in their relatively stable ages during 2004 to 2005.

In Figure 10, we demonstrate the prediction error for different numbers of nodes, where the workload at each node is set as 128 hours. We can find that the prediction error remains low. As the number of nodes increases, we find the prediction error getting smaller. Moreover, when the number of nodes is 512, we obtain a prediction error less than 1%. We can conclude our model is stable with the increase of the number of nodes. Figure 11 illustrates the prediction error for different workload at each node, where the number of nodes is set as 128. We observed that with the increase of workload, the prediction error decreases.

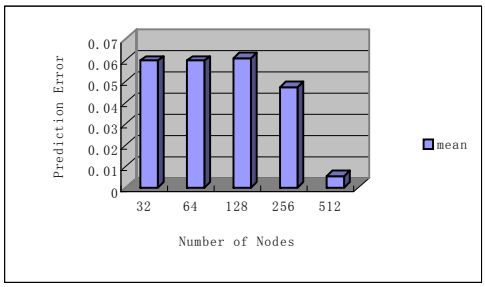

Figure 10. Prediction error with different machine numbers using failure data from LANL systems

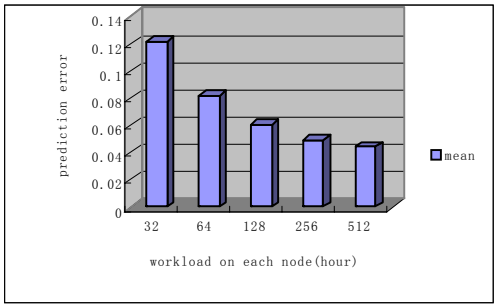

Figure 11. Prediction error with different workloads on each node using failure data from LANL systems

Simulations are conducted to test the efficiency of the proposed fault-aware scheduling algorithms. We use lognormal distribution to characterize the resource failure downtime, which is the best fit for the empirical CDF of downtime in the data [15]. We compare the performance of five scheduling strategies: fault-aware, random, speed-only, reliability-only, and speed-reliability:

- Fault-aware: the proposed heuristic fault-aware scheduling algorithms given in Figure 7.
- Random: machines are randomly selected for task scheduling.
- Speed-only: machines are selected for task allocation based on their speeds. For example, when the number of parallel processes is 10, the first 10 fastest machines are selected for task allocation in speed-only scheduling.
- Reliability-only: machines are selected for task allocation based on reliability, which is defined to be the probability that the machine is not failed during execution.
- Speed-reliability: The selection criterion is the production of speed and reliability.

In this experiment, we set the system size as 100 and 1000 respectively. Each node has a different computing power and a heterogeneous failure arrival and repair pattern. The fastest machine runs 5 times faster than the slowest machine. The failure rate range and the downtime range are the same as the above simulation. At each system size, we schedule a parallel task with different processes numbers. Table 1 gives the average application execution time (hours) over 30 simulations for each scheduling algorithm. We can see from Table 1 that the proposed fault-aware scheduling has the lowest application completion time among the scheduling algorithms in all tests. Straightforward fault-sensitive scheduling algorithms, such as the speed-reliability algorithm, may not work well. The superiority of the fault-aware scheduling is due to the modeling presented in Section 2. An interesting phenomenon observed in the experiment is that when the size of parallel processing (the number of parallel processes) is small, the second best scheduling is the speed-only scheduling instead of speed-reliability. This is because failure arrival and repair has little effect on application performance in a small-scale system. Simply using the production of speed and reliability as the resource selection criterion may exaggerate the impact of failure on application performance, thus leading to an inappropriate decision. However, as the size of parallel processing increase (200 for example), the speed-reliability scheduling outperforms the speed-only scheduling.

Table 1. Application execution time with different scheduling strategies

Scheduling Methodology	Application Execution Time (hours)					
	100 nodes			1000 nodes		
	10	20	40	100	200	400
Fault-aware	**314.7**	**207.8**	**168.4**	**370.2**	**256.0**	**228.5**
Random	612.0	351.6	221.3	695.4	451.9	372.3
Speed-only	362.5	239.4	182.8	486.7	348.8	287.1
Reliability-only	472.2	277.2	185.1	545.7	339.6	262.4
Speed-reliability	449.0	275.6	184.3	524.2	332.8	259.5

6. Related work

Understanding the impact of failures and fault tolerant mechanisms is an active research area. Young gives a simple model to analyze the expected total lost time due to failure and checkpointing. However, his model does not consider the failure repair time on application performance. Duda has studied the impact of both checkpointing and failure repair on application performance [4]. A limitation of his model is the assumption that the failure repair time following exponential distribution, where exponential distribution is a poor fit for fair repair time [15]. Recently, Garg and Huang proposed a checkpointing model to minimize the completion time of a program assuming a general distribution of failure arrival [5]. As the Young and Duda's models, Garg and Huang's model only provides the mean of application completion time of a sequential program. The variation and cumulative density function of the application completion time are not identified in the above three models, which make it impossible to apply these models for parallel processing. The assumption about the exponential repair time in Duda's model is later relaxed by Nicola, Kulkarni and Trivedi [11] to a general distribution by using a "structure-state" process to describe the transition among different failure states, and the Laplace-Stieltjes transform of the application execution time subject to failures is derived. The application of their models in practice, however, is elusive due to the challenge of obtaining the semi-Markov process of failure states, which also prevents the analysis of checkpointing mechanism on the application performance. Moreover, their models focus on the impact of failures on the system performance instead of individual

applications. In contrast, the model proposed in this study is based on probability analysis and rigorous simulations [6]. The derived formulas in Section 2 are not only simple, but also easily applicable in practice for both sequential and parallel computing.

Conventional task scheduling in distributed computing traditionally do not consider system failure. Recently, some work has been done in allocating task considering reliability, where reliability is defined to be the probability that none of the system components fails during execution. In [16], Srinivasan and Jha proposed a heuristic allocation algorithm to maximize the task allocation reliability. Dogan and Ozguner proposed two cost functions in task scheduling to introduce the reliability of resources into decision making [3], in which a reliable task-machine pair is given a higher priority in task scheduling. The potential resource failure probability has been considered in task scheduling in BlueGene/L systems [12]. These works demonstrate the significance of fault-aware task scheduling. However, their scheduling strategies usually aim to improve the system utilization/throughput at the presence of faults instead of individual application performance. In contrast, our proposed fault-aware scheduling algorithms improve the long-running application performance under failure. In addition, our scheduling decision is based on the impact of failure and the corresponding failure handling mechanisms (checkpointing) on the application execution time while their work is based on the application performance loss due to failure, where the impact of failure handling on application performance is ignored. They assume that a resource remains in the failed state for the rest of application execution once it fails.

7. Conclusions and Future Work

System failure has become an important factor of high-end computing. In this research, through a systematic study, we present a solution to improve performance under failure of high-end computing. The solution is twofold, design optimal fault tolerant environment and design optimal task scheduling. We first introduce a performance model describing the effect of system failure on the application completion time. The impact of machine computing power, local failure pattern, and parallel task allocation on the application completion time are individually identified. Next, we extend our model further for a specific failure handling strategy where checkpointing is periodically performed. Based on the proposed models, we then derive performance prediction formulas which provide the mean, variation, and CDF of application completion time under failure in sequential and parallel processing. These performance predictions provide a foundation for design optimal fault tolerant systems and fault-aware task scheduling. Finally, some optimal and heuristic fault-aware scheduling algorithms are developed based on the theoretical foundation.

Intensive simulation and experimental testing are conducted using actual failure trace of high-end computers. The measured experimental results match the analytical prediction closely. The prediction error is kept less than 10% for a test consisting of different values of different design factors. Conventional task scheduling algorithms do not consider the failure/recover factor. Our experimental results show that the improvement of the newly proposed fault-aware scheduling is significant for large-scale high-end computing systems, where failure rate is high.

This research of performance under failure is aimed to large-scale parallel and distributed computing. It has a real potential for the upcoming petaflop high-end computers and may have an immediate impact on Grid computing. In Grid computing, or other service oriented computing, a server that fails to deliver an agreed service on time may not be due to a hardware or software failure. It may be due to other factors, such as policy change and local user interruption, etc. Therefore, the failure rate is high and performance under failure is an immediate issue. In the future, we plan to extend the current checkpointing-based application performance model to other fault tolerant mechanisms such as process migration and program restart. We are interested in integrating the proposed failure models and task scheduling algorithms with existing failure measurement and handling mechanisms to develop an automated failure handling system in parallel and distributed environments. We will collect failure data in Grid and other distributed computing environments, and further test the newly proposed fault-aware scheduling on both parallel and distributed environments.

8. Acknowledgements

We would like to thank Dr. Bianca Schroeder and Dr. Garth A. Gibson at Carnegie Mellon University for their help in analysis of the failure data collected from production systems at Los Alamos National Laboratory. This research was supported in part by National Science Foundation under NSF grant EIA-0224377, CNS-0406328, CNS0509118, and CCF-0621435. Fermi National Laboratory is operated by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

9. References

[1] Berman, F., Wolski, R., Casanova, H., Cirne, W., et al. "Adaptive Computing on the Grid Using AppLeS," *IEEE Transactions on Parallel and Distributed Systems*. Vol 14, No4, pp. 369-382, 2003.

[2] Dinda, P., and O'Hallaron, D. "Host load prediction using linear models," *Cluster Computing*, Vol 3, pp. 265-280, 2000.

[3] Dogan, A., and Ozguner, F. "Reliable matching and scheduling of precedence-constrained tasks in heterogeneous distributed computing," *In Proc. of the 29th International Conference on Parallel Processing*, pp. 307-314, Toronto, Canada, Aug., 2000.

[4] Duda, A. "The Effects of Checkpointing on Program Execution Time," *Information Processing Letters*, vol. 16, pp. 221-229, June 1983.

[5] Garg, S., Huang, Y., Kintala, C., and Trivedi, K. S., "Minimizing Completion Time of a Program by Checkpointing and Rejuvenation," *In Proc .of 1996 ACM SIGMETRICS Conference*, pp. 252-261, Philadelphia, PA, May 1996.

[6] Gong, L., Sun, X-H., and Waston, E. "Performance Modeling and Prediction of Non-Dedicated Network Computing," *IEEE Trans. on Computers*, Vol 51, No 9, pp. 1041-1055, Sep., 2002.

[7] Gross, D., Harris, C. M., *Fundamentals of Queuing Theory*, 3rd Edition, John Wiley & Sons, 1998.

[8] Los Alamos National Laboratory, Operational Data to Support and Enable Computer Science Research, *http://institute.lanl.gov/data/lanldata.shtml*

[9] Lu, Charng-da "Scalable Diskless Checkpointing for Large Parallel Systems," Ph.D dissertation, Department of Computer Science, University of Illinois at Urbana-Champaign, 2005.

[10] Milojicic, D. S., Douglas, F., Paindaveine, Y., Wheeler, R., and Zhou, S. "Process Migration," *ACM Computing Surveys*, Volume 32, No 3, Sep., 2000.

[11] Nicola, V. F., Kulkarni, V. G., and Trivedi, K. S. "Queueing Analysis of Fault-Tolerant Computer Systems," IEEE Trans. Software Engineering, Vol. SE-13, No. 3, pp. 363-375, 1987.

[12] Oliner, A., Sahoo, R. K., Moreira, J. E., Gupta, M., and Sivasubramaniam, A. "Fault-Aware Job Scheduling for BlueGene/L Systems," *in Proc. of the 18th IEEE International Parallel and Distributed Processing Symposiu*m , Santa Fe, New Mexico, Apr., 2004.

[13] Oliner, A., Sahoo, P. K., Moreira, J.E., and Gupta M., "Performance Implications of Periodic Checkpointing on Large-scale Cluster Systems," *in Proc. of the 19th IEEE International Parallel and Distributed Processing Symposium*, Denver, Colorado, Apr., 2005.

[14] Pradhan, D.K. *Fault-Tolerant Computer System Design*, Prentice Hall, Inc., 1996.

[15] Schroeder, B., and Gibson, G.A. "A large-scale study of failures in high-performance computing systems," *in Proc. of the 2006 InternationalConferenceon Dependable Systems and Networks*, Philadelphia, PA, June 2006.

[16] Srinivasan, S.., and Jha, N.K. "Safety and Reliability Driven Task Allocation in Distributed Systems," *IEEE Trans. Parallel and Distributed Systems*, Vol 10, No 3, pp. 238-251, 1999.

[17] Wolski, R. "Dynamically forecasting network performance using the network weather service," *Cluster Computing*, Vol 1, pp. 119-132, 1998.

[18] Wu, M., and Sun, X.-H. "Grid Harvest Service: A Performance System of Grid Computing," *Journal of Parallel and Distributed Computing*, Vol. 66, No. 10, pp. 1322-1337, 2006.

[19] Young, J. W. "A First Order Approximation to the Optimal Checkpoint Interval," *Comm. ACM*, Vol. 17, No 9, pp. 530-531, 1974.

10. Appendix:

Proof of Formula 3

$$\begin{aligned}E(T) &= E(E(T \mid S)) \\ &= E(w + Y_1 + Z_1 + Y_2 + Z_2 + \ldots + Y_S + Z_S \mid S) \\ &= E(w + SE(Y_1) + SE(Z_1)) \\ &= (\frac{1}{1-\lambda_f \mu_f} + \lambda_f \mu_c) w\end{aligned}$$

Proof of Formula 4

$$\begin{aligned}V(T) &= E(V(T \mid S)) + V(E(T \mid S)) \\ &= E(V(w + U(S) \mid S)) + V(E(w + U(S) \mid S) \\ &= E(SV(Y_1) + SV(Z_1)) + V(w + SE(Y_1) + SE(Z_1)) \\ &= \lambda_f w V(Y_1) + \lambda_f w V(Z_1) + \lambda_f w(E^2(Y_1) + E^2(Z_1) \\ &\quad + 2E(Y_1)E(Z_1)) \\ &= \lambda_f w E(Y_1^2) + \lambda_f w E(Z_1^2) + 2\lambda_f w E(Y_1)E(Z_1) \\ &= (\frac{\mu_f^2 + \sigma_f^2}{(1-\lambda_f \mu_f)^3} + \mu_c^2 + \sigma_c^2 + 2\frac{\mu_f \mu_c}{1-\lambda_f \mu_f}) \lambda_f w\end{aligned}$$

Proof of Formula 6 and 7

Since

$$E(T) = E(T \mid S > 0)\Pr(S > 0) + E(T \mid S = 0)\Pr(S = 0)$$

and

$$V(T) = V(T \mid S > 0)\Pr(S > 0) + V(T \mid S = 0)\Pr(S = 0)$$

,

we can get

$$E(T \mid S > 0) = \frac{E(T) - we^{-\lambda_f w}}{1 - e^{-\lambda_f w}},$$

$$V(T \mid S > 0) = \frac{V(T)}{1 - e^{-\lambda_f w}}.$$

Thus, we have the mean and variance of $U(S)$ given $S > 0$

$$\begin{aligned}E(U(S) \mid S > 0) &= E(T \mid S > 0) - w \\ &= \frac{\frac{\lambda_f \mu_f}{1-\lambda_f \mu_f} + \lambda_f \mu_c}{1 - e^{-\lambda_f w}} w\end{aligned}$$

$$\begin{aligned}V(U(S) \mid S > 0) &= V(T \mid S > 0) \\ &= (\frac{\mu_f^2 + \sigma_f^2}{(1-\lambda_f \mu_f)^3} + \mu_c^2 + \sigma_c^2 + 2\frac{\mu_f \mu_c}{1-\lambda_f \mu_f}) \frac{\lambda_f w}{1 - e^{-\lambda_f w}}\end{aligned}$$

Proof of formula (14)

Let b denotes the expected subtask completion time. Using formula (3), we have,

$$\begin{aligned}b &= (\frac{1}{1-\lambda_{f,k}\mu_{f.k}} + \lambda_{f,k}\mu_{c,k}) \frac{w_k}{\tau_k} \\ &= (\frac{1 + \rho_{c,k} - \rho_{c,k}\rho_{f,k}}{1 - \rho_{f,k}}) \frac{w_k}{\tau_k}\end{aligned}$$

Because $w = \sum_{k=1}^{q} w_k = \sum_{k=1}^{q} b\tau_k \left(\frac{1-\rho_{f,k}}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}}\right)$,

we can get

$$b = \frac{w}{\sum_{k=1}^{q} \tau_k \left(\frac{1-\rho_{f,k}}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}}\right)}.$$

Thus, the subtask workload w_k is calculated as

$$w_k = \frac{\left(\frac{1-\rho_{f,k}}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}}\right)\tau_k}{\sum_{k=1}^{q} \tau_k \left(\frac{1-\rho_{f,k}}{1+\rho_{c,k}-\rho_{c,k}\rho_{f,k}}\right)} w .$$