Combining Partial Redundancy and Checkpointing for HPC

Kishor Kharbas, David Fiala, Frank Mueller, Kurt Ferreira, Christian Engelmann
North Carolina State University, Sandia National Laboratories, Oak Ridge National Laboratories
mueller@cs.ncsu.edu

Abstract

Today's largest High Performance Computing (HPC) systems exceed one Petaflops (10^{15} floating point operations per second) and exascale systems are projected within seven years. But reliability is becoming one of the major challenges faced by exascale computing. With billion-core parallelism, the mean time to failure is projected to be in the range of minutes or hours instead of days. Failures are becoming the norm rather than the exception during execution of HPC applications.

Current fault tolerance techniques in HPC focus on reactive ways to mitigate faults, namely via checkpoint and restart (C/R). Apart from storage overheads, C/R-based fault recovery comes at an additional cost in terms of application performance because normal execution is disrupted when checkpoints are taken. Studies have shown that applications running at a large scale spend more than 50% of their total time saving checkpoints, restarting and redoing lost work.

Redundancy is another fault tolerance technique, which employs redundant processes performing the same task. If a process fails, a replica of it can take over its execution. Thus, redundant copies can decrease the overall failure rate. The downside of redundancy is that extra resources are required and there is an additional overhead on communication and synchronization.

This work contributes a model and analyzes the benefit of C/R in coordination with redundancy at different degrees to minimize the total wallclock time and resources utilization of HPC applications. We further conduct experiments with an implementation of redundancy within the MPI layer on a cluster. Our experimental results confirm the benefit of partial, dual and triple redundancy and show a close fit to the model. We show that combined C/R and redundancy results in shorter overall execution time even for medium-sized HPC applications with 4,000 processes and partial redundancy (a replica for every other process). At 60,000 processes, dual redundancy requires twice the number of processing resources for an application but allows two jobs of 128 hours wallclock time to finish within the time of just one job without redundancy. For other configurations, partial redundancy results in the lowest time. Partial redundancy further allows one to trade-off additional resource requirements for redundancy against wallclock time, which provides a tuning knob for users to adapt to resource availabilities.

1. Introduction

Today's HPC systems are commonly utilized by long-running application jobs that employ MPI message passing as an execution model [16, 35]. Yet, application execution may be interrupted by faults. For large-scale HPC, faults have become the norm rather than the exception for parallel computation on clusters with 10s/100s of thousands of cores. Past reports attribute the causes to hardware (I/O, memory, processor, power supply, switch failure etc.) as well as software (operating system, runtime, unscheduled maintenance interruption). In fact, recent work indicates that (i) servers tend to crash twice a year (2-4% failure rate) [32], (ii) 1-5% of disk drives die per year [27] and (iii) DRAM errors occur in 2% of all DIMMs per year [32], which is more frequent than commonly believed.

Even for small systems, such causes result in fairly low mean-time-between-failures/interrupts (MTBF/I) as depicted in Table 1, and the 6.9 hours estimated by Livermore National Lab for its BlueGene confirms this. In response, long-running applications on HEC installations are required to support the checkpoint/restart (C/R) paradigm to react to faults. This is particularly critical for large-scale jobs: As the core count increases, so does the overhead for C/R, and it does so at an exponential rate. This does not come as a surprise as any single component failure suffices to interrupt a job. As we add system components (such as cores, memory and disks), the probability of failure combinatorially explodes.

System	# CPUs	MTBF/I
ASCI Q	8,192	6.5 hrs
ASCI White	8,192	5/40 hrs ('01/'03)
PSC Lemieux	3,016	9.7 hrs
Google	15,000	20 reboots/day
ASC BG/L	212,992	6.9 hrs (LLNL est.)

Table 1. Reliability of HPC Clusters [17]

For example, a study from 2005 by Los Alamos National Laboratory estimates the MTBF, extrapolating from current system performance [26], to be 1.25 hours on a Petaflop machine. The wall-clock time of a 100-hour job in such as system was estimated to increase to 251 hours due to the C/R overhead implying that 60% of cycles are spent on C/R alone, as reported in the same study. More recent investigations [7, 8] revealed that C/R efficiency, *i.e.*, the ratio of useful vs. scheduled machine time, can be as high as 85% and as low as 55% on current-generation HEC systems.

# Nodes	work	checkpt	recomp.	restart
100	96%	1%	3%	0%
1,000	92%	7%	1%	0%
10,000	75%	15%	6%	4%
100,000	35%	20%	10%	35%

Table 2. 168-hour Job, 5 year MTBF

A study by Sandia National Lab from 2009 [14] shows rapidly decaying useful work for increasing node counts (see Table 2). Only 35% of the work is due to computation for a 168 hour job on 100k nodes with a MTBF of 5 years while the remainder is spent on checkpointing, restarting and then partial recomputation of the work lost since the last checkpoint. Table 3 shows that for longer-running jobs or shorter MTBF (closer to the ones reported above), useful work becomes *insignificant* as most of the time is spent on restarts.

job work	MTBF	work	checkpt	recomp.	restart
168 hrs	5 yrs	35%	20%	10%	35%
700 hrs	5 yrs	38%	18%	9%	43%
5,000 hrs	1 yr	5%	5%	5%	85%

Table 3. 100k Node Job, varied MTBF

The most important finding of the Sandia study is that **redundancy in computing can significantly revert this picture.** By doubling up the compute nodes so that every node N has a shadow node N', a failure of primary node N no longer stalls progress as the shadow node N' can take over its responsibilities. Their prototype, rMPI, provides dual redundancy [14]. And *redundancy scales*: As more nodes are added to the system, the probability for simultaneous failure of a primary N *and* its shadow rapidly decreases. This is due to the fact that only one node of the remaining n-1 nodes after a failure represents the shadow node, and only failing this node causes the job to fail — and choosing just that shadow node becomes less likely as the number of nodes increases (see the "birthday problem" in Section 4 for details). Of the above overheads, the recompute and restart overheads can be nearly eliminated (to about

1%) with only the checkpointing overhead remaining — at the cost of having to deploy twice the number of nodes (200,000 nodes in Table 3) and up to four times the number of messages [14]. But once restart and rework overheads exceed 50%, redundancy is actually *cheaper* than traditional C/R at large core counts.

In summary, redundancy cuts down the failure rate of the MPI application, which result in less overhead for checkpointing and repeated execution. The downside is that additional computing resources are required depending on the degree of redundancy, i.e, dual (2x), triple (3x) or some partial level of redundancy (1.5x, 2.5x). There is also an increase in the total execution time due to redundant communication.

Contributions: In this work, we try to answer following questions: (1) Is it advantageous to use both C/R and redundancy at the same time to improve performance or job throughput? (2) What are the optimal values for the (partial) degree of redundancy and checkpoint interval to achieve the best performance?

HPC users, depending on their needs, may have different goals. The primary goal of the user may be to complete application execution in the smallest amount of time. Other users may want to execute their application with least number of required resources. A user may also create a cost function giving different weights to execution time and number of resources used.

We derive a mathematical model to analyze the effect of using both redundancy and checkpointing on the execution time of the application. Using this model, we identify the best configuration to optimize the cost of executing the application. Our results not only show benefits due to redundancy for large-scale clusters, they also provide a model to fine-tune application needs: Given a set of spare nodes, a HPC job can exploit partial redundancy where only some nodes are replicated. This delimits the resource requirements of redundancy at a lower reliability level. Thus, partial redundancy presents a knob to trade off cost vs. benefit.

2. Background

A widely researched topic in HPC is to mitigate the effects of faults occurring during the execution of an application. Individual faults are generally classified into permanent, transient, and intermittent [1]. A permanent fault is a fault that continues to exist until it is repaired. A transient fault is a fault that occurs for a finite time and disappears at an unknown frequency. Intermittent faults occur and disappear at a known frequency.

A parallel HPC system is composed of a number of system components with processes that cooperate to solve a single job. The system components (nodes) are coupled via communication so that failure of one process can lead to failure of the entire job. Process failure is often classified into one of the following categories [10]:

(1) Fail-stop failure: when the process completely stops, e.g., due to system crash. (2) Omission failure: if a process fails to send or receive messages correctly. (3) Byzantine failure: when a process continues operating but propagates erroneous messages or data. Byzantine failures are usually caused by soft errors resulting from radiation.

In this work, we focus on the issues emanating from fail-stop process failures. Detection and correction of Byzantine errors using software redundancy and voting are beyond the scope of this paper.

Fault tolerance uses protective techniques to provide a resilient computing environment in the presence of failures. These techniques can be broadly classified into Algorithm-Based Fault Tolerance (ABFT), message logging, checkpoint/restart and replication/redundancy. ABFT requires special algorithms that are able to adapt to and recover from process loss due to faults [19]. ABFT is achieved by techniques such as data encoding and algorithm redesign. MPI-based ABFT applications require resilient message passing. E.g., FT-MPI [12] continues the MPI task even if some processes are lost. Applications that follow a master/slave programming paradigm can be easily adapted to ABFT applications [22].

Message logging techniques record message events in a log that can be replayed to recover a failed process from its intermediate state. All message logging techniques require the application to adhere to the piecewise deterministic assumption that states that the state of a process is determined by its initial state and by the sequence of messages delivered to it [30].

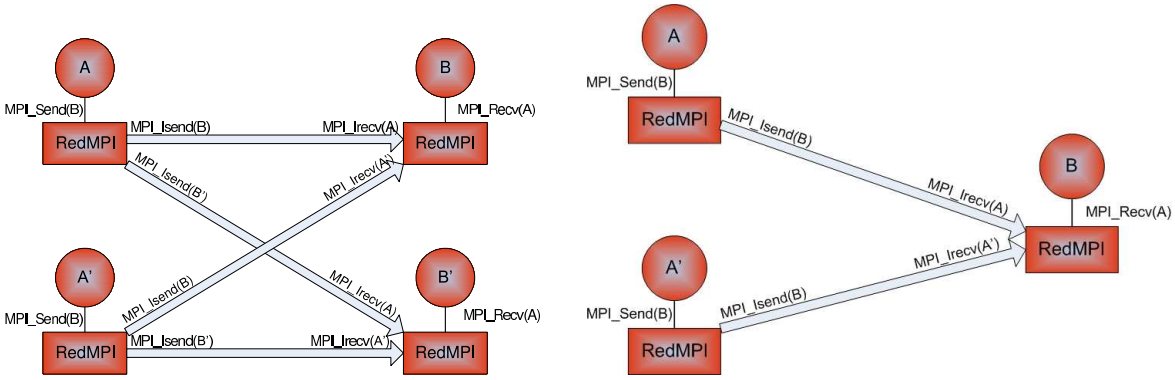
Checkpoint/restart (C/R) techniques involve taking snapshots of the application during failure-free operation in a synchronous fashion and storing them to stable storage. Upon failure, an application is restarted from the last successful checkpoint. Stable storage is an abstraction for some storage devices that ensure that the recovery data persists through failures.

Checkpoint Restart: The C/R service supported by MPI runtime environments utilizes a single-process checkpoint service specified by the user as a plug-in facility. Depending upon the transparency with regard to the application program, single-process checkpoint techniques can be classified as application level, user level or system level. *Application-level* checkpoint services interact directly with the application to capture the state of the program [33]. *User-level* checkpoint services are implemented in user space but are transparent to the user application. This is achieved by virtualizing all system calls to the kernel, without being tied to a particular kernel [24]. *System-level* checkpointing services are either implemented inside the kernel or as a kernel module [9]. The checkpoint images in system-level checkpointing are not portable across different kernels. for our experiments, we used Berkeley Lab Checkpoint Restart (BLCR) [9], a system-level checkpoint service implemented as a Linux kernel module.

The state of a distributed application is composed of states of each individual process and all the communication channels. Checkpoint coordination protocols ensure that the states of the communication channels are consistent across all the processes to create a recoverable state of the distributed application. These protocol events are triggered before the individual process checkpoints are taken. A distributed snapshot algorithm [4], also commonly known as Chandy-Lamport algorithm, is one of the widely used coordination protocols. This protocol requires every process to wait for marker tokens from every other process. After a process receives tokens from every other process, it indicates that the communication channel between the process and every other process is consistent. At this point, this process can be checkpointed. The checkpoint coordination protocol implemented in OpenMPI [20] is an all-to-all bookmark exchange protocol. Processes exchange message totals between all peers and wait until the totals equalize. The Point-to-point Management Layer (PML) in OpenMPI tracks all messages moving in and out of the point-to-point stack.

As expected, the C/R techniques come at an additional cost since performing checkpoints interrupts the normal execution of the application processes. Additional overhead is incurred due to sharing of processors, I/O storage, network resources, etc. When assessing the cost of C/R fault tolerance techniques, we must consider the effect on both the application and the system. Checkpoint overhead accounts for the increase in execution of the application due to the introduction of a checkpoint operation [28, 34]. Checkpoint latency is the time required to create and establish a checkpoint to a stable storage. Various optimization techniques have been studied to improve both forms of overhead as described below.

Forked checkpointing forks a child process before the checkpoint operation is performed [5, 33]. The image of the child process is taken, while the parent process resumes execution. Afterward, the pages that have changed since the fork are captured from the parent process, thereby reducing the checkpoint overhead.



(a) Redundancy (b) Partial Redundancy

Figure 1. Blocking Point to Point Communication

Checkpoint compression is a method for reducing the checkpoint latency by reducing the size of process images before writing them to stable storage [23, 33]. *Memory exclusion* skips temporary or unused buffers to improve checkpoint latency [29]. This is done by providing an interface to the application to specify regions of memory that can be safely excluded from the checkpoint [2]. *Incremental checkpointing* reduces the checkpoint latency by saving only the changes made by the application from the last checkpoint. These techniques commonly rely on hardware paging support, e.g., the modified or dirty bit of the MMU [15, 18]. During recovery, incremental checkpoints are combined with the last full one to create a complete process image.

Redundancy: To decrease the failure rate for large-scale applications, redundancy can be employed at the process level [14, 21]. Multiple copies (or replicas) of a process run simultaneously, so that if a process stops performing its desired function, a replica of the process can take over its computation. Thus, a distributed application can sustain failure of a process if redundant copies are available. An active node and its redundant partners form a node sphere that is considered to fail if all the nodes in the sphere become non-functional. Overall, redundancy increases the mean time between failures (MTBF). This allows us to checkpoint less frequently while retaining the same resiliency level.

rMPI [14] developed at Sandia National Laboratories is a user-level library that allows MPI applications to transparently use redundant processes. MR-MPI [11] is a modulo-redundant MPI solution for transparently executing HPC applications in a redundant fashion that uses the PMPI layer of MPI. VolpexMPI [21] is a MPI library implemented from scratch that supports redundancy internally with the objective to convert idle PCs into virtual clusters for executing parallel applications. In Volpex MPI, communication follows the pull model; the sending processes buffer data objects locally and receiving process contact one of the replicas of the sending process to get the data object. RedMPI is another user-level library that uses the PMPI layer to implement wrappers around MPI calls and provide transparent redundancy capabilities. RedMPI is capable of detecting corrupt messages from MPI processes that become faulted during execution. With triple redundancy, it can vote out the corrupt message and thereby provide the error-free message to the application. The library operates in one of the two modes: All-to-all mode or Msg-PlusHash mode. In All-to-all mode, complete MPI messages are sent from each replica process of the sender and received by each replica of the receiver process. In contrast, one complete MPI message from a sender replica and a hash of the message from another replica is received by the receiver process in Msg-PlusHash mode. We used the RedMPI redundancy library with its All-to-all mode in this work for experiments.

3. Design

The RedMPI library is positioned between the MPI application and the standard MPI library (e.g., OpenMPI, MPICH2). It is implemented inside the profiling layer of MPI and intercepts all the MPI library calls made by the application. No change is needed in the application source code. The redundancy module is activated by MPI_Init(), which divides the MPI_COMM_WORLD communicator into active and redundant nodes.

To maintain synchronization between the redundant processes, each replica should receive exactly the same messages in the same order. This is performed by sending/receiving MPI messages to/from all the replicas of the receiver/sender process.

Consider the scenario shown in Figure 1(a), where A sends a message via MPI_Send() to process B while Process B issues a blocking receive operation via MPI_Recv(). Process A has 2 replicas, A and A', similarly process B has 2 replicas, B and B'. Corresponding to this send operation, process A performs a non-blocking send to each of the replicas of the destination process, B and B'. Only after both these sends have been completed is the send performed by the application considered complete. The redundant partner of A, A', performs exactly the same operations.

At the receiver side, process B posts two receive calls, one receive from A and other from A'. In the general case, a process posts receives from all redundant partners of the sender processes that are alive. The RedMPI library allocates additional buffers for receiving redundant copies. When all receives are complete, the message from one of the buffers is copied to the application-specified buffer before returning control to the application.

Figure 1(b) depicts the sequence of steps that take place when partial redundancy is employed. Here, process A has two replicas while process B has just one. Hence, process B receives two messages via two MPI_Recv() calls. On the other hand, processes A and A' send just one message each to the single copy of process B.

Special consideration is required for wildcard receives (MPI_ANY_SOURC
Since a message sent from any process can complete a wildcard receive request, we have to make sure that all the replicas of the process get the message from the same "virtual" sender. To ensure this, RedMPI performs the following steps:

(1) On the receiving node B, only one receive operation with tag MPI_ANY_SOURCE is posted.

(2) When this call completes, the receiver information is determined and is sent to node B' (if it exists). Also, another receive is posted to determine the message from the remaining replicas of the sender process.

(3) Node B' uses this envelope information to post a specific receive call to obtain the redundant message from the node that sent the first message to B.

The MPI specification requires that non-blocking MPI calls return a handle to the caller for tracking the status of the corresponding communication. When redundancy is employed, corresponding to a non-blocking MPI call posted by the application, multiple non-blocking MPI calls are posted for each replica of the peer process. RedMPI maintains the set of request handles returned by all the non-blocking MPI calls. A request handle that acts as an identifier to this set is returned to the user application. When the application, at a later point, issues a call to MPI_Wait(), RedMPI waits on all the requests belonging to the set before returning from the call.

4. Mathematical Analysis

We make the following assumptions in our model about the execution environment: (1) Studies [31] have shown that the failure rate of a system grows proportional to the number of sockets in the system. Researchers usually consider a socket as a unit of failure and refer to the number of sockets when measuring system reliability. But for simplicity and to abstract away machine specific details from the discussion, we refer to *nodes* in this work. Here, a *node* refers to an execution unit that fails independently. Most commonly, the term *node* is used interchangeably with *socket*. (2) Each process of the parallel application is allotted a separate node. Spare resources are used for performing redundant computation. This means that if an application running N processes moves to 2x redundancy, it now utilizes $2N$ processes on twice the number of nodes. This assumption guarantees that redundancy does not slow down the computation of the application. (3) Node failures follow a Poisson process. The interval between failures is given by an exponential distribution. (4) The failure model is that of fail-stop behavior. This is the most frequent failure type in practice that can be detected via timeout-based monitoring. Other failure models are beyond the scope. (5) Spare nodes are readily available to replace to failed node. This gives an implied assumption that failures can occur anytime between the start and the end of application execution, i.e., failures can occur even when a checkpoint is taken or when the application is restarted after a failure.

4.1 Degree of Redundancy and Reliability

When redundancy is employed, each participating application process is a sphere of replica processes that perform exactly the same task. The replicas coordinate with each other so that another copy can readily continue their task after failure of a copy. The set (sphere) of replica processes, performing the same task and hidden from each other at the application level, is called a virtual process. The processes inside a sphere are called "physical processes".

Here, we present a qualitative model of the effect of redundancy on reliability of the system. Reliability of a system is defined as the probability that the given system will perform its required function under specified conditions for a specified period of time.

The analysis that follows applies not only to MPI-based applications but to any parallel applications where failure of one or more participating processes cause failure of the entire application.

Consider such a parallel application with the following parameters:

N: number of virtual processes involved in the parallel application;

M: number of virtual messages within the parallel application;

r: redundancy degree (number of physical processes per virtual process);

$N \times r$: total number of physical processes;

t: base execution time of the application;

θ: Mean Time to Failure (MTBF) of each node.

(1) As discussed before, due to the overhead of redundancy, the time taken by the application due to redundancy is greater than the base execution time. The overhead depends on many factors, including communication to computation ratio of the application, degree of redundancy, placement of replicas and relative speed of the replica processes. It is very difficult to construct an exact formula to represent the overhead in terms of the degree of replication. In the analysis developed here, we consider overhead due to redundant communication but ignore overheads caused by other factors, such as redundant I/O (which is not supported by RedMPI and not triggered in experiments).

Let α be the communication/computation ratio of the application. Hence, $(1-\alpha)$ is the fraction of the original time t required for computation. This time remains the same with redundancy since only communication is affected by redundancy. The remaining time, namely $\alpha \cdot t$, is the time required for communication, which is affected by redundancy.

All collective communications in MPI are based on point-to-point MPI messages (except when hardware collectives are used). The redundancy library interposes the point-to-point calls and sends/receives to/from each copy of the virtual process. Thus, each point-to-point MPI call is translated into r point-to-point MPI calls per physical process, where r is the redundancy level (e.g., 2 or 3).

Hence, the total number of point-to-point MPI calls per process with redundancy is r times the number of MPI point-to-point calls per process in plain (non-redundant) execution. This implies that the total communication time with redundancy is $r \cdot \alpha \cdot t$. The total execution time with redundancy can then be expressed as

$$t_{Red} = (1-\alpha)t + \alpha t r. \quad (1)$$

(2) As per the assumption, the arrival of failures for each node follows a Poisson process. Hence, reliability of a physical process, which is the probability that the process survives for time t, is $R(t) = e^{-t/\theta}$. When θ is large, it can be approximated as $R(t) = 1 - t/\theta$. Hence, the probability that a node survives for the entire duration t_{Red} of application execution is $P(Survival) = 1 - t_{Red}/\theta$. Using this result, the probability of failure of a node over the time period t_{Red} is $P(Failure) = 1-(1-t_{Red}/\theta) = t_{Red}/\theta$.

(3) Each virtual process has r physical processes, implying the following relation:
Reliability of a virtual process
= P(at least one physical process survives)
= 1 - P(all physical processes fail)
$= 1 - (t_{Red}/\theta)^r$.

(4) Since failure of one or more virtual processes will make the entire application fail, all N virtual processes need to survive until the end. Thus, the reliability of the entire system is:
R_{sys} = P(all virtual nodes survive until the end)
$= [1 - (t_{Red}/\theta)^r]^N$.

(5) Reliability can be written in terms of failure rate (λ) as $R(t) = e^{-\lambda t}$. Using the above equation, the failure rate of the system can be obtained as $\lambda_{sys} = -log(R_{sys})/t_{Red}$ or:

$$\lambda_{sys} = -N\frac{log[1-(\frac{t_{Red}}{\theta})^r]}{t_{Red}} \quad (2)$$

Figure 2 shows how varying r (the degree of redundancy) changes the reliability of the entire system, given the indicated sample input parameters,

4.2 Effect of Checkpointing on Execution Time

Checkpointing does not affect the reliability of the system, i.e., it does not improve the mean time between failures, but it avoids the need to restart the process from the beginning by capturing the state of the application at an intermediate execution state.

As discussed earlier, performing checkpoints comes at a cost. Each checkpoint taken has certain overheads depending on various parameters including the number of parallel tasks, time taken to synchronize the processes and time taken to store checkpoints to stable storage. The minimum number of checkpoints should be performed so as to reduce these overheads.

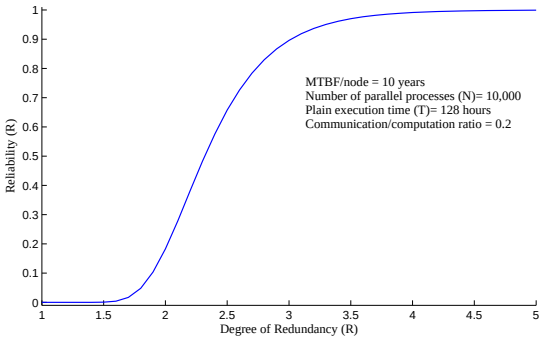


Figure 2. Effect of Redundancy on Reliability

Another consideration while choosing a checkpoint interval is that it determines the average time for repeated execution after a failure. The greater the checkpoint interval, the more rework needs to be performed after a failure to return the application to the state at which a failure occurred.

Consider an application running with the following parameters:

t: time taken by the original application to complete in absence of failure;

λ: system failure rate, i.e., the number of failures per unit time;

δ: checkpoint interval, i.e. the time between successive checkpoints;

c: time required for a single checkpoint to complete;

Θ: mean time between failures of the entire system on which the parallel application runs — can be expressed in terms of failure rate as $1/\lambda$;

R: restart overhead accounting for the time taken to read checkpoint images, instantiation of each application process, coordination between processes, etc.

T_{total}: total time taken for completion, i.e., time after which 't' amount of actual work is performed.

Figure 3 shows the lifetime of a process in the presence of periodic checkpointing and occurrence of failures.

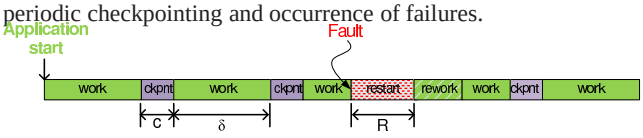


Figure 3. Life-cycle of an application

Number of failures: Failures can occur anytime during the execution of the application, including the restart and rework phases. Hence, the entire execution time of the application, T_{total}, is susceptible to failures. Let n_f be the number of failures that occur till the application completes, which can be calculated as:

$$n_f = \text{Total time} \times \text{Failure rate} = T_{total}\lambda$$

The total time taken by application (T) is the sum of:

(1) The time taken by the application to perform actual computation = t.

(2) The total time taken to take checkpoints until the end of task. We assume that there is no information available about impending failures through a monitoring or feedback system. Instead, we periodically checkpoint at a constant interval of δ. This time is equal to (number of checkpoints) $\times$ (overhead per checkpoint) or:

$$\text{(Total Time)/(Checkpoint interval)} \times c = (t/\delta)c$$

(3) The total restart time derived from the total number of failures during the lifetime of the application and the expected amount of restart time. Since failures can occur even when the application is undergoing restart, the average time spend in a single restart phase is less than the maximum possible time, R, of a restart phase.

(4) The total rework time, i.e., the total time spend on recomputing lost work. The amount of lost work depends on the time at which a failure occurs after a checkpoint is established and also the likelihood of failure during checkpointing (handling multiple failures, incl. failures during recovery). We will use the term t_{lw} to denote the expected amount of work lost due to failures. t_{lw} is also the maximum possible duration of the rework phase.

However, when the application begins the rework phase, there is a possibility that a failure occurs before entirely re-computing the lost work. Thus, the expected time spent in a rework phase is less than the maximum possible time of a rework phase (which is equal to the average amount of lost work).

Since after occurrence of a failure, restart is always followed by rework, we can combine them into a single phase. The maximum duration of this phase is $R + t_{lw}$. Using the same argument as before, the expected time of this combined phase is less than the maximum. Let us denote the expected time of this phase as t_{RR}.

Before deriving t_{RR}, let us find the average amount of work that could be lost due to failure during computation. Let t_{lw} be this expected value, i.e, the expected time at which a failure occurs after a checkpoint is taken.

The computation time of the application can be divided into segments of length $\delta + c$. Each segment consists of work phase (length=δ) followed by checkpoint phase (length=c). The lost work depends on the time at which a failure occurs after the start of a segment. Let $\delta_c = \delta + c$. The PDF describing the probability of failure occurring at time t from the start of a segment can be calculated as:

$$p(t) = \frac{1}{\Theta}e^{\frac{-t}{\Theta}} + \frac{1}{\Theta}e^{\frac{-(t+\delta_c)}{\Theta}} + \frac{1}{\Theta}e^{\frac{-(t+2\delta_c)}{\Theta}} + \cdots = \frac{e^{\frac{-t}{\Theta}}}{\Theta(1-e^{\frac{-\delta_c}{\Theta}})}$$

When a failure occurs at time $0 \leq t \leq \delta$, the lost work is also t. When failure occurs at time $\delta < t \leq \delta_c$ (during checkpoint), lost work is δ. Hence, the expected time for lost work can be calculated as:

$$t_{lw} = \int_0^{\delta} t.p(t)\,\mathrm{d}t + \int_{\delta}^{\delta_c} \delta \cdot p(t)\,\mathrm{d}t$$

Solving the above integral and substituting $\delta_c = \delta + c$ yields:

$$t_{lw} = \frac{\Theta e^{\frac{\delta}{\Theta}} - \delta - \Theta}{e^{\frac{\delta}{\Theta}} - e^{\frac{c}{\Theta}}} + \delta\frac{(1-e^{\frac{-c}{\Theta}})}{e^{\frac{\delta}{\Theta}} - e^{\frac{-c}{\Theta}}}$$

After occurrence of a failure, the application begins in the restart phase, which takes R time followed by t_{lw} rework time. As mentioned earlier, we combine these two phases into a single phase of maximum duration: $R + t_{lw}$. The derivation of the expected time of this phase, t_{RR}, is presented below:

The reliability of a system, i.e, the probability of survival until time t is $e^{\frac{-t}{\Theta}}$. This implies that the probability of a system failing before time $R + t_{lw}$ is:

$$1 - P(\text{system survives up to time } R + t_{lw}) = 1 - e^{\frac{-(R+t_{lw})}{\Theta}}.$$

This also implies that probability of failure after time $(R + t_{lw})$ is $e^{\frac{-(R+t_{lw})}{\Theta}}$. This is the probability that application completes restart and t_{lw} amount of rework. t_{RR} can now be calculated as:

$$\begin{aligned} t_{RR} &= P(\text{failure before } R + t_{lw}) \\ &\quad * (\text{expected time of failure in interval 0 to } R + t_{lw}) \\ &\quad + P(\text{failure after } R + t_{lw}) * (R + t_{lw}) \\ &= (1 - e^{\frac{-(R+t_{lw})}{\Theta}})(\int_0^{R+t_{rw}} t \cdot \frac{1}{\Theta}e^{\frac{-t}{\Theta}}\,\mathrm{d}t) + e^{\frac{-(R+t_{lw})}{\Theta}} \cdot (R + t_{lw}) \\ &= (1 - e^{\frac{-(R+t_{lw})}{\Theta}})(M - (R + t_{lw})e^{\frac{-(R+t_{lw})}{\Theta}} - \Theta e^{\frac{-(R+t_{lw})}{\Theta}}) + \\ &\quad e^{\frac{-(R+t_{lw})}{\Theta}} \cdot (R + t_{lw}). \end{aligned}$$

Thus, the total time spent in rework and restart during the entire run of the application is:

$$\text{(Number of failures)} \times t_{RR} = T_{total}\lambda t_{RR}.$$

The total time taken by the application can be written as:

$$T_{total} = t + \frac{tc}{\delta} + T_{total}\lambda t_{RR}$$

$$T_{total} = \frac{t + \frac{tc}{\delta}}{(1 - \lambda t_{RR})} \quad (3)$$

where

$$t_{RR} = (1 - e^{\frac{-t_{lw}}{\Theta}})(\Theta - t_{lw}e^{\frac{t_{lw}}{\Theta}} - \Theta e^{\frac{-t_{lw}}{\Theta}}) + e^{\frac{-t_{lw}}{\Theta}} \cdot t_{lw} \quad (4)$$

$$t_{lw} = \frac{\Theta e^{\frac{\delta}{\Theta}} - \delta - \Theta}{e^{\frac{\delta}{\Theta}} - e^{\frac{c}{\Theta}}} + \delta\Theta e^{\frac{-\delta}{\Theta}}(1 - e^{\frac{-c}{\Theta}})$$

Its easy to understand that there is a trade-off between the interval between checkpoints, i.e, δ and the checkpoint/restart overhead. Too low a checkpoint interval leads to unnecessary checkpoints and thus higher overheads. On the other hand, having a very high checkpoint interval leads to greater loss of computation due to failures. Thus, we need to choose the right checkpoint interval that gives the minimum overhead.

We qualitatively and quantitatively compared the above equation for total application execution time with the one derived by Daly in [6]. Eq. 3 and Daly's exact formulation are approximated by Eq. 5 below, i.e., the share the same trends (see [6]). We also performed quantitative simulations for the relevant value ranges of the parameters of both Eqs. 3 and 5 to confirm this in plots (omitted due to space). Plots are nearly a perfect match (with single digit % deviations). The next section will feature model and experiment side-by-side (Figure 8).

We observe that these two equations are very similar with a low % difference. Instead of deriving our own optimum checkpoint interval, and to simplify the analysis, we use Daly's optimal checkpoint interval [6]:

$$\delta_{opt} = \sqrt{(2\delta\Theta)}[1 + \frac{1}{3}(\frac{\delta}{2\Theta})^{\frac{1}{2}} + \frac{1}{9}(\frac{\delta}{2\Theta})] - \delta \qquad (5)$$

The minimum time required for application completion can be obtained by substituting δ_{opt} in Eq. 3.

4.3 Combining Redundancy and Checkpointing

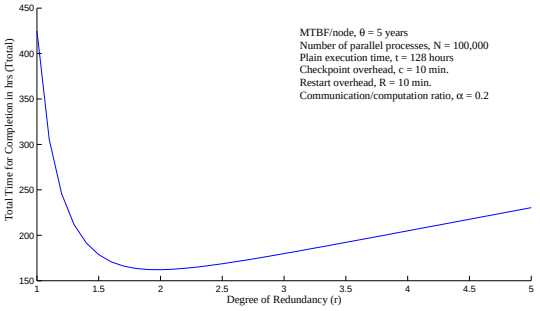


Figure 4. Configuration 1: Total Execution Time with Varying Degree of Redundancy

Employing redundancy helps to increase the reliability of the system. But even a high degree of redundancy does not guarantee failure free execution, though it certainly decreases the probability of failure: The probability of simultaneous failure of a node and its replica is equivalent to the "birthday problem", which is can be approximated as $p(n) \approx 1 - (\frac{n-2}{n})^{n(n-1)/2}$ for n nodes, which very rapidly approaches zero for increasing n, i.e., $\lim_{n\to\infty} p(n) = 0$. Thus, we still need to checkpoint so that an application can be we can recovered after a failure instead of having to re-run the application from the scratch. Eq. 3 shows that the time required for application completion increases as the failure rate increases (equivalently, the MTBF decreases). From Eq. 2, we see that with redundancy we can decrease the failure rate of the system and thus decrease the checkpointing frequency, which ultimately results in less checkpointing overhead and faster execution of the application.

Let us assess the effect of this hybrid approach qualitatively. Below are the set of equations that determine the overall relationship between redundancy and the total time for completion.

(1) The application time with redundancy degree r is $t_{Red} = (1 - a)t + (\alpha t)r$.

(2) The failure rate with redundancy degree r is $\lambda_{sys} = -N\frac{log[1-(\frac{t_{Red}}{\Theta})^r]}{t_{Red}}$.

(3) The total execution time including C/R overheads is $T_{total} = \frac{t_{Red} + \frac{t_{Red}c}{\delta}}{(1-\lambda t_{RR})}$, where t_{RR} is given by Eq. 4.

Figure 4 shows the variation in total time with varying degree of redundancy of an application for an original running time of 128 hrs. We see that as we increase the degree of redundancy, the execution time decreases initially. The minimum time achieved is at 162 hours when the redundancy degree is ≈ 2. As we increase the redundancy further, the total time increases as well. Figure 5 shows the execution time with the same configuration as above, but the MTBF of a node is increased to 15 years. As seen in the graph, the minimum execution time is obtained at a lower redundancy degree of ≈ 1.6.

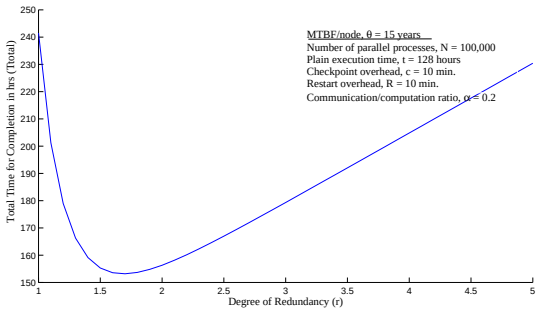


Figure 5. Configuration 2: Total Execution Time with Varying Degree of Redundancy

Figure 6 shows the variation in execution time when the single checkpoint overhead is 1 minute compared to 10 minutes in configuration 1 (Figure 4). The effect of this is that the minimum execution time is obtained at a lower redundancy degree of ≈ 1.7.

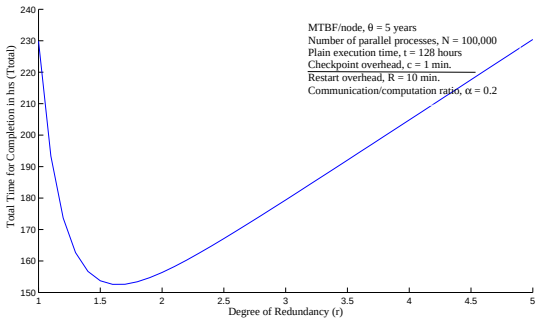


Figure 6. Configuration 3: Total Execution Time with Varying Degree of Redundancy

5. Experimental Framework

Some assumptions and approximations had to be made while performing the mathematical analysis. The most significant one is in Equation 1 and relates to the degree of redundancy and total execution time. Here, we omitted the overheads originating from factors such as placement of replicas and relative speeds of replica processes. It is expected that in a real execution environment the outcome observed will differ to some extent from those in Figures 4, 5 and 6 in the previous section.

To validate the mathematical analysis of the previous section, we collected empirical data by running benchmark applications in a HPC environment. Though the study performed in this work is targeted towards exascale computing, computing systems at such a large scale are not available today. Hence, we run the application to the maximum scale possible on current available resources. Node

failures are injected instead of waiting for actual failures. We scale down the MTBF per node according to the number of nodes available and the execution time of application so that the application suffers a sufficient number of failures to analyze the combined effect of C/R and redundancy. We run the application with a certain degree of redundancy and also checkpoint the application at the optimum frequency calculated from Equation 5. Two processes run in the background of the application, which perform the two tasks specified above. The first background process is the failure injector that triggers failures for the entire application based on per-process failures. The occurrence of a failure for each process is assumed to be a Poisson process.

The failure injector performs the following steps: (1) It maintains a mapping of virtual to physical processes. The status of each physical process at a particular time is either dead or alive. (2) For each physical process in the MPI environment, the time for the next failure is calculated using an exponential distribution that describes the time between events of a Poisson process. (3) As and when the failure time of a physical process is reached, the mapping is updated by marking the process as dead. (4) If all the physical processes corresponding to a virtual process have been marked dead, application termination is triggered followed by a restart from the last checkpoint.

Figure 7 shows how failure of a physical process does not necessarily imply a failure of the MPI application. The application fails and a restart is triggered only when all the physical processes corresponding to a virtual process fail. The second background process is a checkpointer that calculates the optimal checkpoint interval δ using Equations 5 and 2 . It sets a timer for time δ and checkpoints the application when the timer goes off.

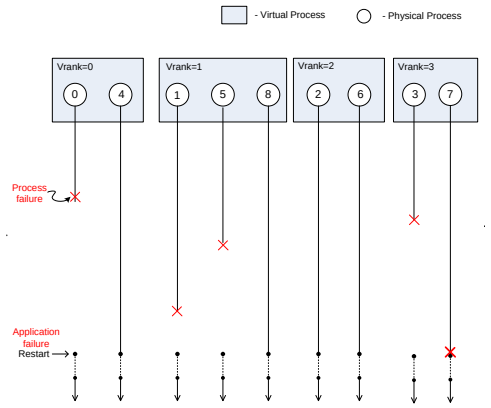


Figure 7. Failure Injection within MPI Applications

6. Results

Experimental platform: Experiments were conducted on a 108 node cluster with QDR Infiniband. Each node is a dual socket shared-memory multiprocessor with two octo-core AMD Opteron 6128 processors (16 cores per nodes). Each node runs CentOS 5.5 Linux x86 64. We used Open MPI 1.5.3 for running our experiments and BLCR as the per-process checkpointing service. The RedMPI library is used for performing redundant computation.

We chose the CG benchmark of the NAS parallel benchmarks (NPB) suite as a test program. CG stands for conjugate gradient. It is used to compute an approximation to the smallest eigenvalue of a large sparse symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long distance communication employing unstructured matrix vector multiplication. CG spends approximately 20% of the total time on MPI communication. Since this study is targeted towards long running applications, we need an application that runs long enough to suffer a sufficient number of failures to assess its behavior in a failure prone environment. Hence, the CG benchmark was modified to run longer by adding more iterations. This was done by repeating the computation performed between MPI_Init() and MPI_Finalize() calls 'n' number of times. This modified CG, class D benchmark with 128 processes takes 46 minutes under failure free execution without redundancy and C/R. Larger inputs would become infeasible (require weeks of experiments). Processes were pinned onto 14 cores per node for application tasks leaving one core each for the operating system and runtime activities.

The MTBF of a node was chosen between 6 hours and 30 hours, with an increment of 6 hours. A MTBF/node of 6 hours gives a high failure rate of $\approx$ 20 failures per hour, while MTBF/node of 30 hours gives a lower failure rate of 4 failures per hour. We ran the application with the injector initially without redundancy and then with double and triple redundancy. To denote redundancy degrees we use the notation "rx" to signify that there are r physical processes corresponding to a virtual process. For example, 2x redundancy means that there are 2 physical processes corresponding to a virtual process. Experiments were also performed with partial redundancy, i.e., some processes have replicas, while some have just one primary copy. For example, a redundancy degree of 1.5x means that every other process (i.e., every even process) has a replica. Partial redundancy was assessed in steps of 0.25x between 1x and 3x.

The results of the experiments for the optimal application execution time using various degrees of redundancy is shown in Table 4. The minimum time taken by the application for each value of MTBF is highlighted in the table. As seen from the results, the minimum application execution time (maximum performance) with MTBF of 6 hours is obtained at 3x redundancy. When the MTBF is 12 hrs, the maximum performance is seen at 2.5x redundancy. Yet for a MTBF of 18, 24 and 30 hrs, the maximum performance is achieved at 2x redundancy. Figures 8 and 9 show these results in the form of line graphs and surface graphs, respectively. As the surface graph shows, local minima exist at different points of the surface indicating an intricate interplay of MTBF and redundancy with respect to overall performance.

MTBF	Degree of Redundancy								
	1x	1.25x	1.5x	1.75x	2x	2.25x	2.5x	2.75x	3x
6 hrs	275	279	212	189	146	158	139	132	***123***
12 hrs	201	207	167	143	103	113	***98***	111	125
18 hrs	184	179	148	120	***72***	126	88	80	84
24 hrs	159	143	133	100	***67***	92	78	84	83
30 hrs	136	128	110	101	***66***	73	80	82	84

Table 4. Performance of an Application (Execution Time in Minutes) with Combined C/R+Redundancy Technique

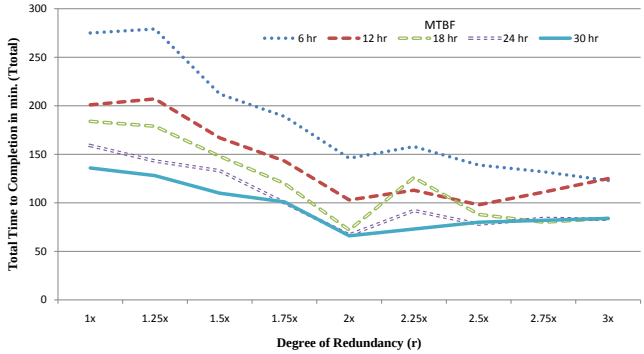


Figure 8. Performance of an Application (Execution Time in Minutes) with Combined C/R+Redundancy Technique on a Cluster

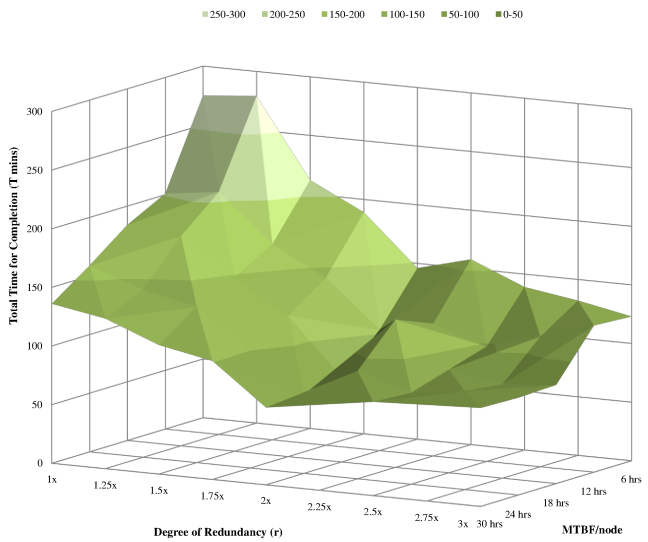


Figure 9. Surface Plot of Performance of an Application (Execution Time in Minutes) with Combined C/R+Redundancy Technique

We make the following observations from the above results: (1) For a high failure rate or, equivalently, lower MTBF (e.g., 6 hrs), the minimum total execution time (maximum performance) is achieved at higher redundancy levels ($>$ 3x in this case).

(2) For lower failure rates (e.g., 24 hrs and 30 hrs) the minimum total execution time (maximum performance) is achieved at a redundancy level of 2x. Increasing the redundancy degree further has adverse effects on execution time.

(3) The minimum execution time (maximum performance) can also be achieved at partial redundancy levels, e.g. for MTBF=12 hrs. Here, the maximum performance is obtained when 2.5x redundancy is employed.

(4) An interesting finding is that in most cases 1.25x redundancy yields poor performance compared to 1x (when no redundancy is employed). Similarly, 2.25x yields poor performance compared to 2x redundancy. This behavior can be attributed to a higher increase in redundancy overhead in return for a smaller decrease in failure rate as we move from 1x to 1.25x (or from 2x to 2.25x). To support this argument, a separate experiment was carried out to calculate the failure-free execution time with increasing redundancy levels. The results are shown in Table 5 and Figure 10. It can seen that the rate of increase in execution time is larger in the first step (i.e., from 1x to 1.25x) while there is a decrease in the rate in the subsequent steps.

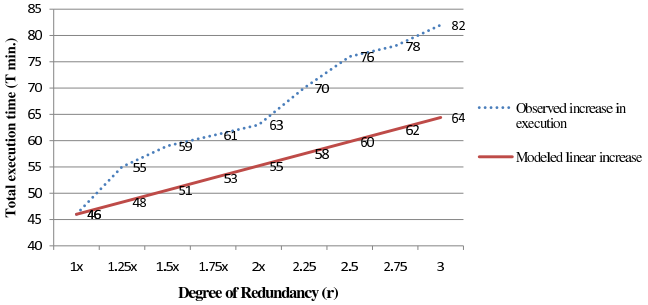


Figure 10. Increase in Execution Time with Redundancy

(5) The purpose of these experiments is to verify the mathematical model developed in Section 4. Hence, we modeled our execution by estimating/calculating the environment parameters and substituting them in the set of equations developed in Section 4.3. There is a subtle difference in the experimental setup and our model discussed in Section 4.3. While running the application, failures are not triggered when a checkpoint is performed or when restart is in progress. Our model, though, considers failures at any time, including checkpointing and restart. We simplified our model to match our experiments, which results in the following time function: $T_{total} = t_{Red} + t_{Red}\sqrt{2c\lambda_{sys}} + t_{Red}\lambda_{sys}R$. We have used this equation for modeling the application behavior in the presence of C/R and redundancy. This simplified model pertains just to this sub-section, specifically to Figures 11 and 12.

The overhead per checkpoint (c) was calculated as 120 sec. by first running the plain application, then running it with one checkpoint taken during execution, and calculating the difference between the later and former execution times. Time taken to restart the application after a failure and beginning of re-execution (restart overhead, R) was measured as approx. 500 sec. The CG benchmark, on average, spends 20% of the total time in MPI communication, so the communication to computation ratio (α) is 0.2. Plotting the equations in MATLAB, we get the expected application behavior shown in Figure 11. It can be seen that the actual behavior of the application (Figure 8) is similar to the modeled behavior shown in Figure 11, thus validating our analytical model. For closer comparison, Figure 12 overlays the performance curves in Figure 11 over those in Figure 8 for selected MTBF values. The trend followed by the observed curves is very similar to the modeled curves. However, we see some absolute differences in the execution times that can be attributed to various causes :

(a) The redundancy overhead in actual runs is more than the modeled overhead (see Figure 10). An increase in the execution time is due to additional failures occurring during this extra time. (b) The fault injector generates failures by using inputs from a random number generator that follow a Poisson distribution. The application running time may not be long enough for the observed failure rate to converge to the average failure rate (λ).

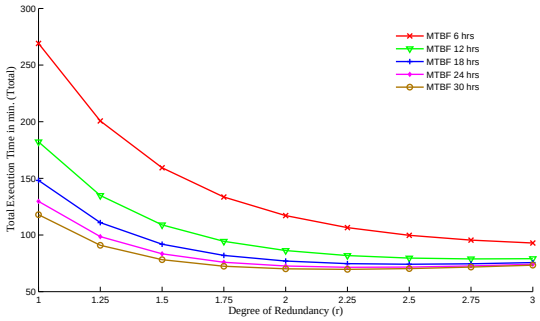


Figure 11. Modeled Application Performance

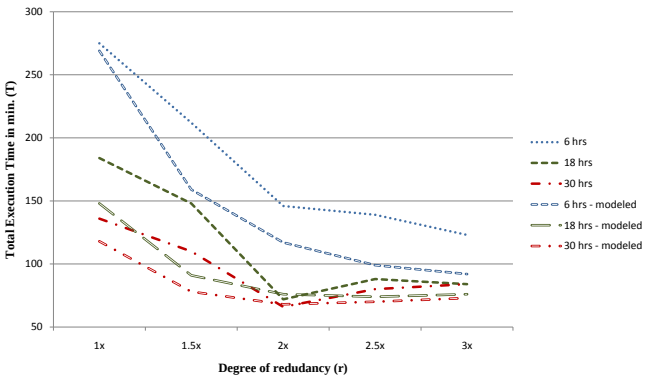


Figure 12. Observed Performance vs. Modeled Performance

Degree of Redundancy	1x	1.25x	1.5x	1.75x	2x	2.25x	2.5x	2.75x	3x
Observed increase in execution time	46	55	59	61	63	70	76	78	82
Expected linear increase	46	48	51	53	55	58	60	62	64

Table 5. Increase in Execution Time with Redundancy

Simulations: We also performed simulations using our analytical model to determine at which point an application begins to benefit from redundancy. Figure 13 depicts the execution time of a 128 hour job for different redundancy levels and number of processes (with a factor of 10,000 on the latter/x-axis) under weak scaling, i.e., the problem size is scaled at the same rate as the number of processes resulting in a constant compute overhead per process. The cross-over points between no redundancy (1x) and dual redundancy (2x) at ≈4,000 processes and triple redundancy (3x) at ≈10,500 processes indicate an early benefit for combined C/R+redundancy. When it is not always feasible to minimize runtime due to resource scarcity, resilience may still be improved through partial redundancy as a tunable knob (e.g., 1.5x). As seen in the figure, when the number of processes is between 3851 and 25180, we obtain the best results by running the application at 1.5x rather than 2x. Only beyond N=25180 does 2x yields a lower execution time than 1.5x.

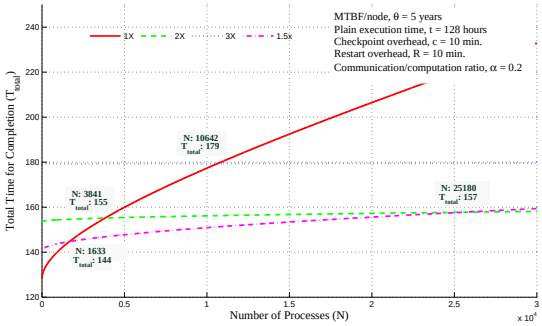


Figure 13. Modeled Wallclock Time of a 128 Hour Job for Different Redundancy Levels up to 30k Nodes

Using additional nodes for redundancy is a cost, while gaining a shorter execution time is a benefit: The nodes become available sooner and can be used for other jobs. Hence, when the runtime with redundancy is twice that of dual redundancy at 60,000 processes, we can actually run two dual redundant jobs of 128 hours in the time of just one job without redundancy (see Figure 14). This indicates that redundancy is a powerful technique to increase the utilization of exascale HPC installation for capacity computing (where job throughput is the objective). It does not provide a solution to capability computing (where all nodes are utilized by a job without redundancy), which presents an open problem to resilience handling of exascale systems. The figure further underlines that pure C/R without redundancy results at exponential increases in execution time after ≈ 60,000 nodes.

7. Related Work

Several models to determine the optimal checkpointing strategy for parallel programs have been developed in prior works. Young [37] presented an optimal checkpoint and recovery model and obtained a constant optimal checkpoint interval to reduce the overall execution time. Based on Youngs work, Daly [6] improved the model to an optimal checkpoint placement from a first order to a higher order approximation. These studies establish a cost function for the total execution time and try to minimize the output of the cost function. The results derived are similar to those obtained in Section 4.2. Other work considers those and additional approximations under a variety of failure distributions [3].

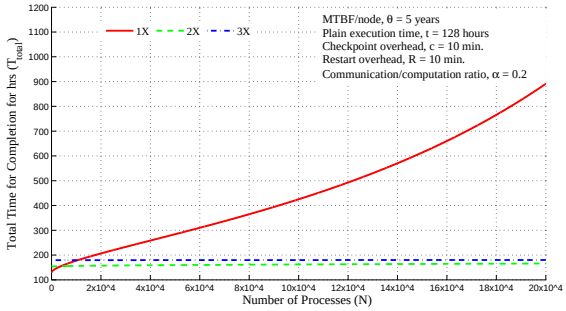


Figure 14. Modeled Wallclock Time of a 128 Hour Job for Different Redundancy Levels up to 200k Nodes

Authors of [29], [36] have taken a different approach by modeling the problem as a Markov availability model and obtained an optimal checkpoint placement that maximizes system availability. [29] has addressed the issue of placing processes on available processors (task mapping) and determining corresponding checkpoint intervals to obtain the best execution time. They model the performance of coordinated checkpointing systems where the number of processors dedicated to the application (termed “a” for active) and the checkpoint interval (termed “I”) are selected by the user before running the program. The model is used to determine the average availability of the program in the presence of failures that can be used to select values of a and I to minimize the expected running time of the program.

In [25], authors have presented a reliability-aware method for an optimal C/R strategy towards minimizing rollback and checkpoint overheads. Their model considers variable checkpoint intervals by taking actual system reliability into account.

The works cited above have considered C/R as the only method for achieving fault tolerance and analyzed the effect of C/R on application execution time. As discussed before, redundancy is another way of achieving fault tolerance. Ferreira et al. [13] have studied the viability of process replication as the primary fault tolerance mechanism for exascale systems, employing C/R as a secondary mechanism. Results from their work show that replication outperforms traditional C/R approaches for large sockets counts and limited I/O bandwidths frequently anticipated at exascale. The study compares only two models of execution, one without redundancy and another with dual (2x) redundancy assuming that processes have to double up on the same number of nodes. In contrast, our work assumes that the number of nodes is increased at the same rate that the number of processes increases under redundancy. This is more realistic since high-performance applications tend to fully utilize the available memory space of a node. Furthermore, we model the execution of an application in the presence of redundancy at various degrees (including partial redundancy) in combination with C/R. Using this model, we study the trade-off between levels of redundancy and checkpoint frequencies with the goal of optimizing system performance.

8. Conclusion

Petascale and forthcoming exascale computing systems experience outages due to failed components, software bugs, and power disruptions. A common method to allow application runs longer than interval between faults is to checkpoint applications to stable storage. But studies show that large-scale applications spend more than 50%

of their time in C/R activities. Another way to achieve fault tolerance is to employ redundancy, wherein multiple processes perform the same computation.

This work shows that C/R-based fault tolerance can be used in synergy with redundancy to optimize application performance. We have developed an analytical model to estimate the execution time of long-running large-scale programs in presence of failures that combines C/R with redundancy. Using this model, HPC users can configure their application with the right amount of redundancy degree and checkpoint frequency to obtain the maximum performance from the available resources. We also validated the model by injecting faults into applications with an implemented redundancy layer on our computing cluster. The modeled application behavior closely mimics the observed application behavior on our cluster and we obtain the maximum performance at the same redundancy levels as given by the model. We observed that there are some deviations from the modeled performance curve, especially at partial redundancies. The reason for such behavior was traced to the deviation of observed redundancy overhead from the expected overhead.

Overall, combined C/R and redundancy results in shorter overall execution time even for medium-sized HPC applications with 4,000 and 25,000 processes for 1.5x and 2x redundancy. At 60,000 processes, dual redundancy (2x) requires twice the number of processing resources for an application but allows two jobs of 128 hours wallclock time to finish within the time of just one job without redundancy. Partial redundancy of 2.5x also results in the lowest time for certain MTBF values. But partial redundancy goes one step further: It allows a trade-off between additional resources and wallclock time, which effectively presents a tuning knob for users to adapt to resource availabilities.

References

[1] Mostafa Abd-El-Barr. Design and analysis of reliable and fault-tolerant computer systems. In *Imperial College Press*, 2007.

[2] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the 18th annual international conference on Supercomputing*, 2004.

[3] Marin Bougeret, Henri Casanova, Mikael Rabie, Yves Robert, and Frédéric Vivien. Checkpointing strategies for parallel jobs. In *Supercomputing*, nov 2011.

[4] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3:63–75, February 1985.

[5] Antonio Cunei and Jan Vitek. A new approach to real-time checkpointing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments*, 2006.

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

[7] John T. Daly. ADTSC nuclear weapons highlights: Facilitating high-throughput ASC calculations. Technical Report LALP-07-041, Los Alamos National Laboratory, Los Alamos, NM, USA, June 2007.

[8] John T. Daly, Lori A. Pritchett-Sheats, and Sarah E. Michalak. Application MTTFE vs. platform MTTF: A fresh perspective on system reliability and application throughput for computations at scale. In *Proceedings of the Workshop on Resiliency in High Performance Computing (Resilience) 2008*, pages 19–22, May 2008.

[9] Jason Duell. The design and implementation of berkeley labs linux checkpoint/restart. Technical report, 2003.

[10] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. In *Journal of the ACM, 35(2):288323*, 1988.

[11] Christian Engelmann and Swen Böhm. Redundant execution of HPC applications with MR-MPI. In *Proceedings of the* 10^{th} *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011*, pages 31–38, February 15-17, 2011.

[12] Graham E. Fagg, Edgar Gabriel, Zizhong Chen, Thara Angskun, George Bosilca, Jelena Pjesivac-Grbovic, and Jack J. Dongarra. Process fault-tolerance: Semantics, design and applications for high performance computing. In *International Journal for High Performance Applications and Supercomputing, 19(4):465478*, 2005.

[13] Kurt Ferreira, Jon Stearley, James H. Laros III, Ron Oldfield, Kevin Pedretti, Ron Brightwell, Rolf Riesen, Patrick Bridges, and Dorian Arnold. Evaluating the viability of process replication reliability for exascale systems. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis, SC'11*, nov 2011.

[14] Kurt B. Ferreira, Rolf Riesen, Ron Oldfield, Jon Stearley, James Laros, Kevin Pedretty, Todd Kordenbrock, and Ron Brightwell. Increasing fault resiliency in a message-passing environment. TR SAND2009-6753, Sandia National Lab, October 2009.

[15] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers. In *SC*, 2005.

[16] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.

[17] Chung-H. Hsu and Wu-C. Feng. A power-aware run-time system for high-performance computing. In *Supercomputing*, 2005.

[18] Shang-Te Hsu and Ruei-Chuan Chang. Continuous checkpointing: joining the checkpointing with virtual memory paging. *Softw. Pract. Exper.*, 27(9):1103–1120, 1997.

[19] Kuang-Hua Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. In *IEEE Transactions on Computers, 33(6):518528*, 1984.

[20] Joshua Hursey. *Coordinated Checkpoint/Restart Process Fault Tolerance for MPI Applications on HPC Systems*. PhD thesis, Indiana University, July 2010.

[21] Troy LeBlanc, Rakhi Anand, Edgar Gabriel, and Jaspal Subhlok. Volpexmpi: an mpi library for execution of. parallel applications on volatile nodes. In *European PVM/MPI Users' Group Meeting*, pages 124–133, September 2009.

[22] Claudia Leopold and Michael Sub. Observations on mpi-2 support for hybrid master/slave applications in dynamic and heterogeneous environments. In *In Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 4192, pages 285292, September*, 2006.

[23] Chung-Chi Jim Li, Elliot M. Stewart, and W. Kent Fuchs. Compiler-assisted full checkpointing. *Software: Practice and Experience*, February 1994.

[24] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the unix kernel. pages 283–290, 1992.

[25] Yudan Liu, R. Nassar, C. Leangsuksun, N. Naksinehaboon, M. Paun, and Stephen Scott. A reliability-aware approach for an optimal checkpoint/restart model in hpc environments. In *Cluster Computing, 2007 IEEE International Conference on*, pages 452 –457, sept. 2007.

[26] Ian Philp. Software failures and the road to a petaflop machine. In *HPCRI: 1st Workshop on High Performance Computing Reliability Issues*. IEEE Computer Society, 2005.

[27] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure trends in a large disk drive population. In *USENIX Conference on File and Storage Technologies*, 2007.

[28] J. S. Plank and W. R. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, pages 48–57, 1998.

[29] James S. Plank, Yuqun Chen, Kai Li, Micah Beck, and Gerry Kingsley. Memory exclusion: Optimizing the performance of checkpointing systems. *Software: Practice and Experience*, February 1999.

[30] Sriram Rao, Lorenzo Alvisi, and Harrick M. Vin. The cost of recovery in message logging protocols. In *IEEE Transactions on Knowledge and Data Engineering, 12(2):160173*, 2000.

[31] Bianca Schroeder and Garth A. Gibson. Understanding failures in petascale computers. *Journal of Physics*, February 2007.

[32] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. Dram errors in the wild: a large-scale field study. In *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 193–204, 2009.

[33] L. M. Silva, J. G. Silva, S. Chapple, and L. Clarke. Portable checkpointing and recovery. In *In Proceedings of the Fourth IEEE International Symposium on High Performance Distributed Computing (HPDC 95), Washington, DC, USA, 1995. IEEE Computer Society*, 1995.

[34] Nitin H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. In *IEEE Transactions on Computers*, 1997.

[35] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel Distributed Computing*, 63(9):853–865, September 2003.

[36] K. Wong and M. Franklin. Distributed computing systems and checkpointing. In *High Performance Distributed Computing, 1993., Proceedings the 2nd International Symposium on*, pages 224 –233, jul 1993.

[37] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, 1974.