ABSTRACT

KHARBAS, KISHOR H. Failure Detection and Partial Redundancy in HPC. (Under the direction of Dr. Frank Mueller.)

To support the ever increasing demand of scientific computations, today's High Performance Computing (HPC) systems have large numbers of computing elements running in parallel. Petascale computers, which are capable of reaching a performance in excess of one PetaFLOPS (10^{15} floating point operations per second), are successfully deployed and used at a number of places. Exascale computers with one thousand times the scale and computing power are projected to become available in less than 10 years. Reliability is one of the major challenges faced by exascale computing. With hundreds of thousands of cores, the mean time to failure is measured in minutes or hours instead of days or months. Failures are bound to happen during execution of HPC applications. Current fault recovery techniques focus on reactive ways to mitigate faults. Central to any kind of fault recovery method is the challenge of detecting faults and propagating this knowledge. The first half of this thesis work contributes to fault detection capabilities at the MPI-level. We propose two principle types of fault detection mechanisms: the first one uses periodic liveness checks while the second one makes on-demand liveness checks. These two techniques are experimentally compared for the overhead imposed on MPI applications.

Checkpoint and restart (CR) recovery is one of the fault recovery methods which is used to reduce the amount of computation that could be lost. The execution state of the application is saved to a stable storage so that after a failure, computation can be restarted from a previous checkpoint rather than from the start of the application. Apart from storage overheads, CR-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, having redundant copies decreases the overall failure rate. The downside of this approach is that extra resources are used and there is an additional overhead of performing redundant communication and synchronization. The objective of the second half of the work is to model and analyze the benefit of using checkpoint/restart in coordination with redundancy at different degrees to minimize the total time and resources required for HPC applications to complete.

Failure Detection and Partial Redundancy in HPC

by
Kishor H. Kharbas

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2011

APPROVED BY:

Dr. Xiaosong Ma

Dr. Xiaohui (Helen) Gu

Dr. Frank Mueller
Chair of Advisory Committee

DEDICATION

To my parents and my sweet sister, Vrushali.

BIOGRAPHY

Kishor was born to Mr. Haridas Kharbas and Mrs. Nirmala Kharbas on May 9, 1986 in Pune district of Maharashtra, India. He was brought up and completed schooling in Ambernath, a small and peaceful suburb of Mumbai. He pursued bachelors degree in Computer Science at University of Mumbai. After completing college in 2007, he worked as a Software Engineer at Tata Consultancy Services (TCS), Bangalore in its financial software division. To gain more in-depth knowledge of Computer Science, he joined the Masters program at NC State in 2009. Since Spring 2010 he has been working in Systems research with Dr. Frank Mueller.

ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Frank Mueller for showing faith in me and providing valuable guidance and support throughout the thesis work. I am very grateful to Dr. Xiaosong Ma and Dr. Xiaohui Gu for serving on my thesis committee. I would also like to thank Dr. Christian Engelmann (ORNL), Dr. Rolf Riesen and Dr. Kurt Ferreira (SNL) for their helpful inputs during weekly meetings. I am thankful to David, Vivek, Shobit, Abhik, and other labmates for assisting me in the course of my research study. Finally, I thank my roommate Gaurang and other friends in Raleigh for their encouragement and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 High Performance Computing

High performance computing (HPC) is the term for an integrated computing environment used to solve large-scale computational problems. Scientists in core fields like physics, chemistry, biology, nuclear engineering, etc., use such powerful computing systems to solve their complex algorithms and run simulations. High performance computing encompasses hardware, software, programming tools and environments and the network interconnect.

Since the growth in performance of uniprocessor systems hit by the frequency wall, HPC systems have shifted from specialized supercomputers to clusters and grids of off-the-shelf microcomputers. Application developers exploit different forms of parallelism found in their application by running independent tasks on different computing elements simultaneously. Today's HPC systems have hundreds of thousands of computing cores. Petascale computing systems, which are capable of reaching performance in excess of one PetaFLOPS (10^{15} floating point operations per second), are successfully deployed and used at a number of places. As of June 2011, there are 7 supercomputers in the world that can achieve performance of more than 1 PetaFLOPS [2]. Exascale computing systems with one thousand times the scale and computing power are also projected to be available in less than 10 years.

HPC applications are designed using the message passing model. This model is characterized by a set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine and/or on arbitrary number of machines. Tasks share data by performing explicit sending and receiving of messages. Data transfer usually requires cooperative operations to be performed by each participating process. For example, a send operation is completed only once a corresponding receive operation is performed.

The Message Passing Interface (MPI) is a specification for developers and users for a message passing API (application programming interface) developed by the MPI forum. The goal of

MPI is to establish a portable, efficient, and flexible standard that will be wildly used for developing message-passing programs [1]. The MPI interface specifications have been developed for Fortran and C/C++ programs. It has now become a de-facto standard for parallel programs running on compute clusters and supercomputers. There are many implementations of the MPI specification, the most popular ones being OpenMPI, MPICH2 and MVAPICH.

1.2 Faults

A widely researched topic in HPC is to mitigate the effects of faults occuring during the execution of an application. A fault is defined as an event in which a component deviates from its normal behavior. A fault can occur due to a physical defect in the component. A system failure is said to occur when the system cannot deliver its intended functionality due to one or more faults in its components. Individual faults are generally classified into permanent, transient, and intermittent [3]. 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 distributed system is composed of a number of systems with processes that cooperate to solve a single job. The systems are usually 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 [14]:

- Fail-stop failure: when the process completely stops, for example due to system crash.
- Omission failure: if a process fails to send or receive messages correctly.
- Byzantine failure: when a process continues operating but propagates erroneous messages or data. Byzantine failures are usually caused by soft errors resulting due to radiation. Soft errors are commonly seen in random access memory (RAM) and are protected with the help of error correcting codes (ECC). Soft errors occurring inside other components including the CPU have not been addressed well. In a distributed system, Byzantine errors are difficult and costly to detect.

In this work, we focus on the issues emanating from fail-stop process failures. Another research project undertaken at the Systems Research Lab at NCSU deals with detection and correction of Byzantine errors using software redundancy and voting.

1.3 Fault Tolerance

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.

ABFT requires special algorithms that are able to adapt to and recover from process loss due to faults [24]. ABFT is achieved by techniques such as data encoding and algorithm redesign. MPI-based ABFT applications require a resilient message passing environment, e.g. FT-MPI [17], that 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 [30]. The master needs to maintain the task unit assigned to the slaves so that it can reassign them if some process stops.

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 [39]. Message logging is usually combined with checkpointing to save time by using message logs for recovery from the previous successful checkpoint.

Checkpoint/restart techniques involve taking snapshots of the application during failure-free operation 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 which ensure that the recovery data persists through failures. Checkpointing an MPI application not only involves checkpointing each participating process but also involves synchronization of the processes and storing information about the MPI environment.

1.4 Hypothesis

Reactive fault tolerance mechanisms rely on the underlying environment to provide failure detection capabilities. Inputs from the failure detection service can be used to trigger reactive measures when failures occur. On the other side of the spectrum, measures to increase the resiliency include checkpoint/restart and redundancy discussed in Chapter 3.

1. We hypothesize that local knowledge of failures is sufficient for triggering reactive fault tolerance for MPI applications. Thus, failure detection can be implemented using low-cost mechanisms as opposed to traditional heartbeat and tree-based algorithms that incur higher overheads.

2. We also hypothesize that redundancy-based fault tolerance can be used in synergy with CR-based fault tolerance to achieve better application performance for large-scale HPC applications than can be achieved by any of the two techniques alone. By modeling the application behavior and running simulations, we can study the behavior of an application at different levels of redundancy and checkpoint intervals.

Chapter 2

Failure Detection

2.1 Introduction

The current road map to exascale computing faces a number of challenges, one of which is reliability. Given the number of computing cores, projected to be as large as a million, with ten of thousands of multi-socket nodes, components are poised to fail during the execution of large jobs due to a decreasing mean time between failures (MTBF) [36, 43]. When faults become the norm rather than the exception, the underlying system needs to provide a resilience layer to tolerate faults. As discussed in Section 1.3, proposed methods for resilience range from transparent techniques, such as checkpointing and computational redundancy, to explicit handling, such as in probabilistic or fault-aware computing. The latter approach requires significant algorithmic changes and is thus best suited for encapsulation into numerical libraries [16]. This work focuses on the former techniques. It builds on recently developed techniques such as checkpointing (with restarts or rollbacks) or redundant computing in high-performance computing [5, 12, 7, 48, 28] or API extensions for checkpointing [16, 35]. A common challenge of transparent resiliency lies in the detection of faults, which is the focus of this work. This work is combined and built upon an independent study done by Kamal KC [27].

2.2 Motivation

Previous work suggested guidelines on the theoretical methodology for designing fault detection services. However, a practical implementation still poses a challenge in terms of completeness and accuracy because of the diversity of parallel system environments in terms of both hardware and software, which exposes a number of complications due to potentially unreliable failure detectors [8].

The fault model of this work is that components are subject to fail-stop behavior. In other words, components either work correctly or do not work at all. Transient or byzantine failures are not considered. A component is an entire compute node or a network connection / link between any two nodes. In such a framework, we base fault detection on timeout monitoring between endpoints. The focus of our work is to study the impact of timeout monitoring on application behavior such as to perturb application performance as little as possible.

Contributions: In this work, we implement a fault detector (FD) over the MPI profiling layer to detect failures of an MPI application. An FD can be included at various layers of the software stack. We chose the MPI communication layer to implement the FD. We detect MPI communication failures and, at the same time, also utilize the MPI layer as a means to implement detection. This approach has the advantage that it does not require any failure detection support from the underlying software/hardware platform.

In this framework, we observe the effect of a failure, such as lack of response for communication between any two nodes due to node or network failures. We do not perform root cause analysis, which is orthogonal to our work. We assume that the system model provides temporal guarantees on communication bounds (sometimes also combined with computation bounds) called "partially synchronous" [42]. The FD utilizes a time-out based detection scheme, namely, a ping-pong based implementation with the following properties:

- Transparency: The FD can be embedded in MPI applications without any additional modification. The FD runs independently with a unique communicator different from an application program. When MPI applications call MPI_Init, the FD module is activated for each MPI task (on each node) as an independent thread through the MPI profiling interposing layer.

- Portability: MPI applications can be compiled without the FD. Applications only need to be re-linked with the profiling layer of MPI and the FT module. It is not necessary for MPI applications to change in their environment, design or source code. The FD works for arbitrary MPI implementations and has been tested with MPICH, Open MPI, and the LAM/MPI-family.

- Scalability: The FD operates in two modes. It can be configured to send a check message sporadically whenever the application has invoked a communication routine. An alternative setting performs periodic checks at configurable intervals.

The rationale behind sporadic and periodic liveness probing is that the former can be designed as low-cost background control messages that are only triggered when an application is causally dependent on other nodes. The latter, in contrast, can be designed independent of any communication pattern but requires constant out-of-band checking but is agnostic of application communication behavior.

The experimental results show that the FD satisfies the above three properties. The results further indicate that the sporadic approach imposes lower bandwidth requirements of the network for control messages and results in a lower frequency of control messages per se. Nonetheless, the periodic FD configuration is shown to result in *equal or better* application performance overall compared to a sporadic liveness test for larger number of nodes, which is a non-obvious result and a contribution to the community. Our resulting implementation can easily be combined with reactive checkpoint/restart frameworks to trigger restarts after components have failed [45, 12, 41, 7, 26, 21, 22, 23, 21, 40, 51, 34, 47, 48, 49].

2.3 Design

In principle, an FD can be designed using a variety of communication overlays to monitor liveness. A traditional heartbeat algorithm imposes high communication overhead in an all-to-all communication pattern with a message complexity $\Theta(n^2)$ and a time complexity of $\Omega(n)$. This overhead can be high, and a single node does not need to inquire about liveness of all other nodes in an MPI application.

A tree-based liveness check results in $\Theta(n)$ messages with a $\Omega(log(n))$ time complexity where the root node has a collective view of liveness properties. However, mid-level failures of the tree easily result in graph partitioning so that entire subtrees may be considered dysfunctional due to the topological mapping of the overlay tree onto a physical network structure (e.g., a multi-dimensional torus).

We have designed two principle types of failure detection mechanisms. First, we support a sporadic (or on-demand) FD. Second, we have devised a periodic, ring-based FD. The periodic FD can be integrated into MPI applications or may function as a stand-alone liveness test separate from MPI applications. These approaches differ in their liveness probing periods and their network overlay structure.

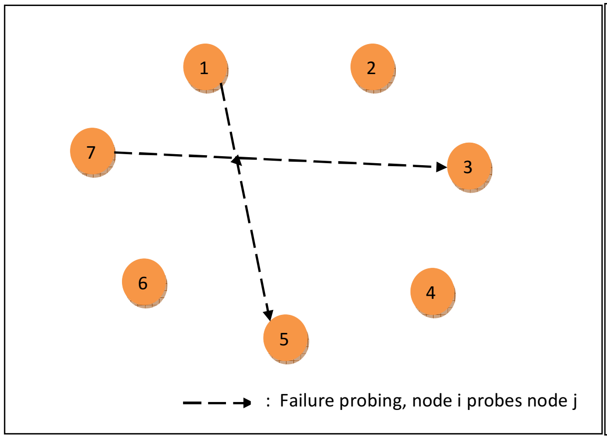
2.3.1 Failure Detector Types

Periodic Ring-Based Failure Detection

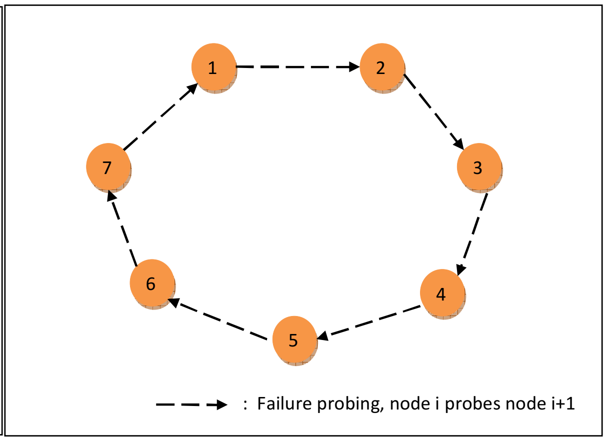
In this approach, starting from initialization of the MPI environment, we form a ring-based network overlay structure wherein the i-th node probes the $(i + 1)$-th node in the ring (see Figure 2.2b). Thus, each node probes its neighbor in the ring irrespective of whether there is any active application communication between the two nodes or not. The probing is performed until the application terminates.

This structure results in $\Theta(n)$ messages for liveness checking and imposes $\mathcal{O}(1)$ time (assuming synchronous checking) or up to $\mathcal{O}(n)$ time (for asynchronous checking that has to propagate around the ring), yet liveness properties are only known about immediate neighbors. For MPI applications, we argue that local knowledge is sufficient to trigger reactive fault tolerance at a higher level.

Figure 2.1: Fault Detection Techniques.



(a) Sporadic Fault Detection.



(b) Periodic Fault Detection.

Sporadic/On-demand Failure Detection

In this approach, a node p probes a node q only if p and q are engaged in an application-level point-to-point message exchange. If p needs to wait beyond a timeout interval for q to resume its work, a control message from p to q is issued (see Figure 2.2a). This happens when node p makes a blocking MPI call, such as MPI_Recv() or MPI_Wait(). Similarly, if the application is engaged in collective communication, such as MPI_Bcast(), and the MPI call does not return within a timeout interval, a ring-based liveness check is triggered. If the liveness test is successful but the application-level MPI call has not been completed, the liveness check is periodically repeated.

This method of liveness check imposes $\mathcal{O}(1)$ message and time overhead, and lifeless properties are only known to immediate neighbors. The control message overhead of this approach may be zero when responses to application messages are received prior to timeout expiration. In such a setting, the overhead is localized to a node and amounts to request queuing and request cancellation (in the best case).

2.4 Implementation

Our implementation assumes that there are reliable upper bounds on processing speeds and message transmission times. If a node fails to respond within a time-out interval, the node is assumed to have failed under this model (fail-stop model). The implementation builds on this assumption when a node starts probing another node. Node pairs are determined by a causal dependency implied from the application communication pattern (for sporadic point-to-point communication) or through network overlays (for sporadic collectives and all periodic liveness checks). Probing is implemented via ping-pong messages monitoring round trip time (RTT) timeouts. Probing for failure detection can be parametrized as follows: (a) INTER-PROBE: This interval determines the frequency of probing, i.e., the time between successive probes by a node. Longer values may cause late detection of failure while shorter intervals allow for early detection but increase the overhead imposed by control messages. (b) TIME-OUT: This interval determines the granularity of failure detection but also impacts the correctness of the approach. If the interval is chosen too small, a slow response may lead to false positives (detection of failure even though the node is functional). Conversely, a large interval may delay detection of failures. Determination of a "good" timeout interval is non-trivial, even if we assume an upper bound on network and processing delay (see above).

We have used the MPI profiling layer to implement one version of the FD module. Wrappers have been written for MPI functions. These wrappers take appropriate FD actions before and after calling the PMPI versions of the corresponding communication function. When the application calls MPI_Init(), a duplicate communicator, DUP_MPI_COMM_WORLD, is created, which is subsequently used to send control messages for failure detection. The software stack of the FD in conjunction with an application is depicted in Figure 2.2. Application-level MPI calls (depicted as Fortran calls) trigger a language-neutral wrapper before calling the interposed MPI function. In the PMPI wrapper, the native MPI function is called (prefixed with PMPI_). The fault detector governs back-channel exchanges of control message over the duplicated communicator. Another version of the FD implements periodic checks as stand-alone processes separate from an MPI application.

The fault detector is implemented as a pair of threads, namely sender and receiver threads. We require MPI to support multi-threading, i.e., MPI_Init_thread() should support MPI_-THREAD_MULTIPLE to ensure thread support. The sender thread triggers an ALIVE message (ping) or waits for an acknowledgment (ACK) message (pong) up to a given timeout. The receiver thread receives ALIVE queries over the new communicator from the sender thread and responds with an ACK message. The failure detection module maintains a queue of events in sorted order of event times. An event could be "sending out a new probe to some node i" or "end of timeout interval for a probe sent to some node i". Upon such an event, the sender

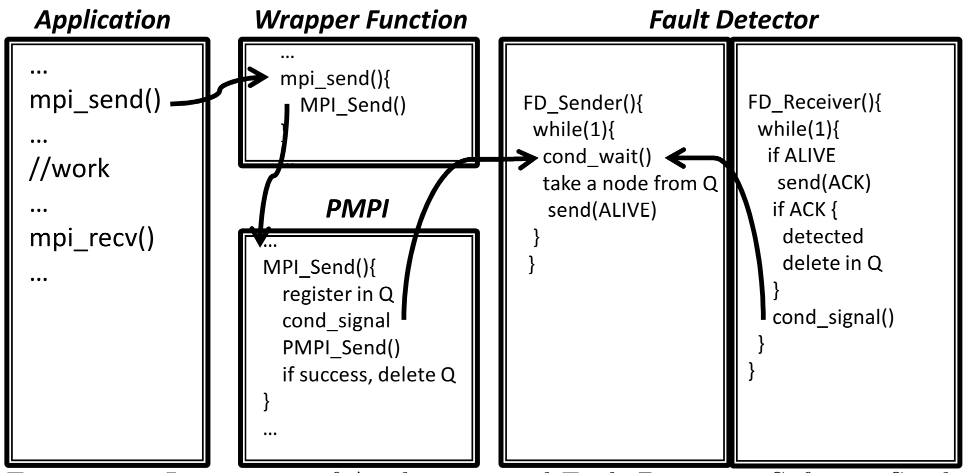


Figure 2.2: Interaction of Application and Fault Detection Software Stacks

thread is activated and performs the respective action.

2.5 Performance Evaluation

We measured the overhead incurred by the FD module for the set of NAS parallel benchmarks (NPB V3.3) with input classes C and D [6]. Using gettimeofday(), wall-clock times of applications were measured between MPI_Init() and MPI_Finalize() calls with and without failure detector interpositioning or backgrounding. Tests were performed by running each configuration five times and computing the average overhead for different inter-probe intervals and number of processes.

2.5.1 Experimental Platform

Experiments were conducted on a 108 node cluster with Infiniband QDR. Each node is a 2-way 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 1.5.1 for evaluating the performance of the FD.

2.5.2 Benchmark Results

Figures 2.5, 2.6, 2.3 and 2.4 depict the relative overheads of fault detection for 64 and 128 processes (over 64 nodes) with periodic and sporadic fault detection, respectively. Overheads of the FD approach for fault tolerance with inter-probe frequencies of 1-10 seconds ("FD 1 sec" to "FD 10 sec") are plotted relative to application execution without fault tolerance ("No FD"), i.e., in the absence of the FD module (normalized to 100%).

We first observe that for 64 process applications both the sporadic and periodic FD have overheads ranging from less than 1% to 14% averaging around 4-5%. We further observe that

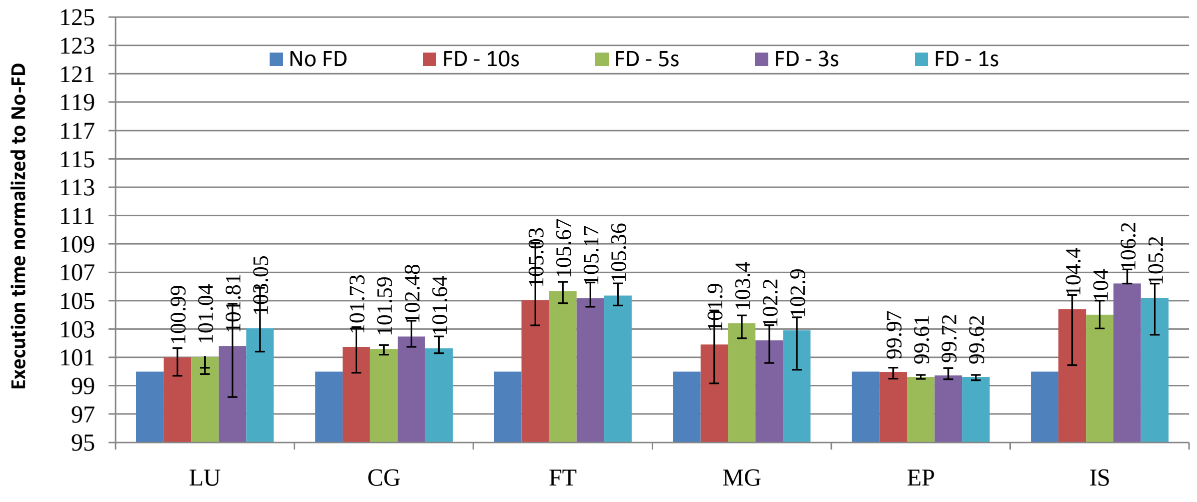


Figure 2.3: Overhead of Periodic Fault Detection for 64 Processes

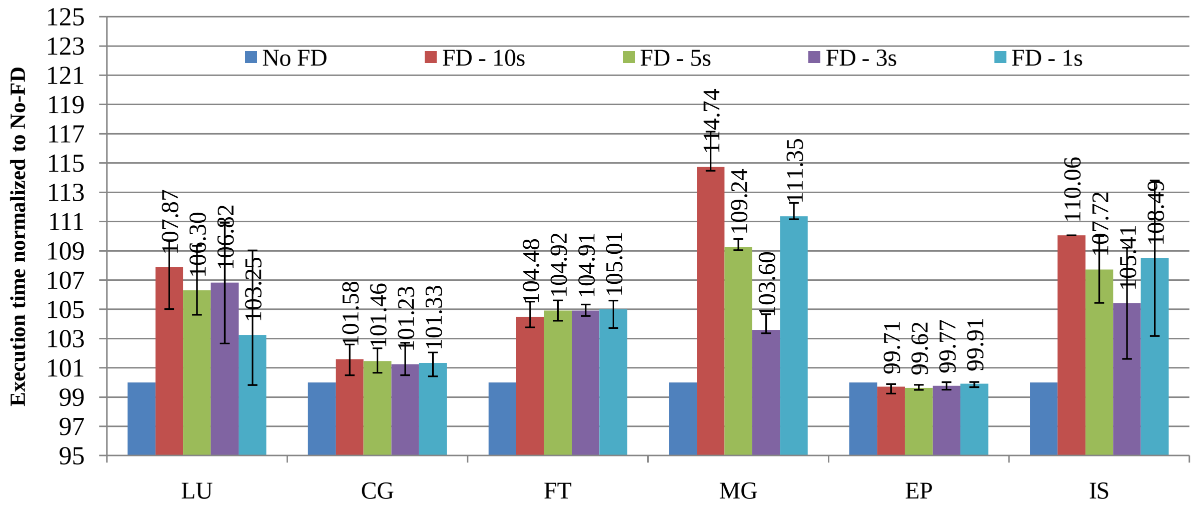


Figure 2.4: Overhead of Sporadic Fault Detection for 64 Processes

periodic either matches or outperforms the sporadic approach by 1-13%. Similarly, for 128 process runs the overheads range from less than 1% to 21% averaging around 10%. Again here, periodic either matches or outperforms sporadic by 2-6%.

These observed results can be explained as follows: As overall communication is increasing, timeouts in the sporadic mode happen more frequently, in particular for collectives where communication results in contention (e.g., for all-to-all collectives). Sporadic control messages only add to this application-induced contention. In contrast, the periodic approach has the advantage that control messages are evenly likely to occur across the entire application duration. This probabilistic spread frequently results in control messages being issued during computation,

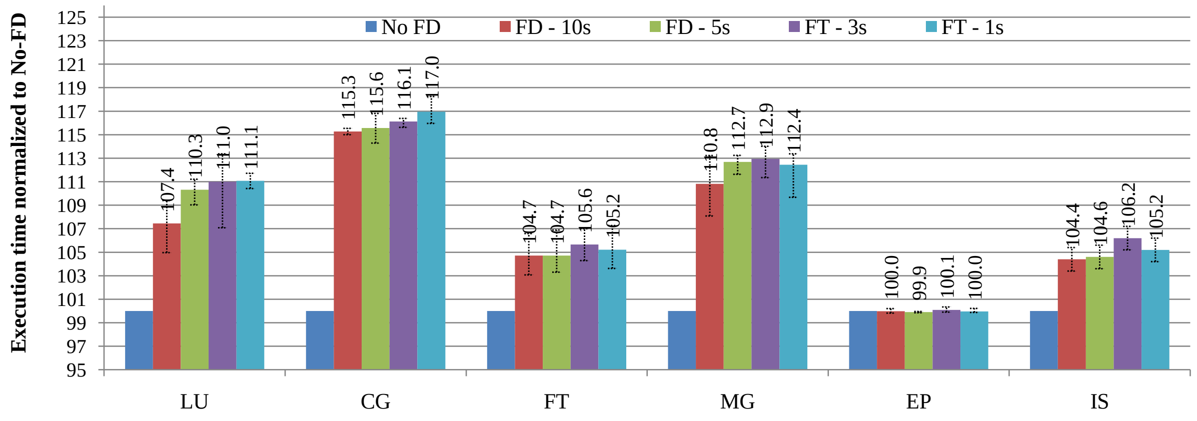


Figure 2.5: Overhead of Periodic Fault Detection for 128 Processes

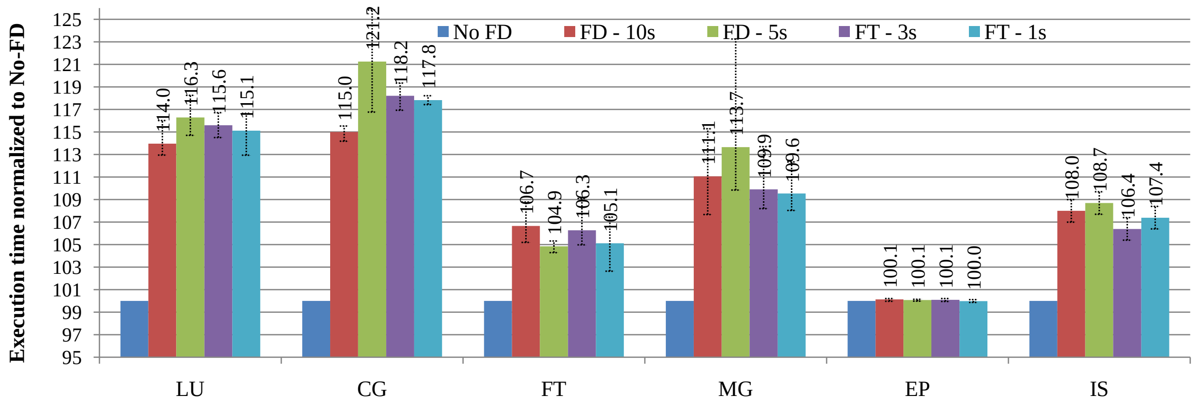


Figure 2.6: Overhead of Sporadic Fault Detection for 128 Processes

i.e., when the network interconnect is not utilized at all. This trend increases with strong scaling (larger number of nodes).

In addition to studying the overhead, we make the following observations based on application benchmark behavior from these results for *sporadic* liveness detection (Figures 2.4 and 2.6).

- The overhead of sporadic fault detection differs significantly for different benchmarks. Each MPI call from the application imposes modifications of the internal control message queues associated with the fault detection module, which adds some processing overhead. Hence, benchmarks with more frequent MPI calls suffer more overhead. For example, the CG, LU and MG benchmarks incur a larger number of total MPI calls and consequently incur higher overheads.
- The value of the inter-probe interval does not have much significance in determining

the overall overhead as most of the calls are completed within the sporadic inter-probe interval.

- As the number of processes increases, so does the communication to computation ratio. Hence, the FD overhead increases as we increase the number of parallel processes. This effect can be seen more clearly in the LU and MG benchmarks where the average overhead of sporadic FD of 128 processes (Figure 2.6) is often larger than that for 64 processes (Figure 2.4).

Figures 2.3 and 2.5 show the performance overhead of periodic liveness probing for 64 and 128 processes, respectively. We make the following observations based on application benchmark behavior from these results for *periodic* liveness detection.

- The overhead of periodic fault detection also varies between different applications. Benchmarks with more frequent communication or larger messages suffer more overheads. For example, LU and CG are more communication intensive and thus cause more network contention between probe message traffic and application traffic, resulting in higher overhead. On the opposite side, benchmarks like EP, FT and IS are less communication intensive and result in less overhead.

- In most cases, the overhead of the FD approach increases when we decrease the inter-probe interval. As probing takes place from start to end, decrease in inter-probe intervals generates a larger number of probes. This increases the processor utilization as well as network bandwidth demand, which results in increasing overheads.

We further conducted experiments with periodic liveness checking as a background activity in separate processes across nodes that an MPI application is running on. The results depicted in Figure 2.7 show absolutely no overhead for NPB codes over 128 processes except for IS with an overhead of 4.5%. We also varied the number of MPI tasks per node and found these results to remain consistent up to 15 tasks per node. Only at 16 tasks per node did overheads spike to up to 28-60% depending on the NPB code (Figure 2.8). This shows that as long as a spare core is available for background activity, the impact of out-of-band communication on application performance is minimal. In HPC, applications tend to utilize only a subset of cores of high-end multi-core nodes as in our case, which ensures that communication does not become a bottleneck.

Overall, the results show that periodic failure detection performs better than sporadic for communication intensive codes and that separation of the FD from MPI applications reduces their perturbation.

2.6 Related Work

Tushar and Sam classify eight classes of failure detectors by specifying completeness and accuracy properties [8]. They further show how to reduce the eight failure detectors to four and discuss how to solve the consensus problem for each class. This work has influenced other contemporary work as it raises the problem of false positives for asynchronous systems. In our work, we focus on single-point failure detection. Consensus is an orthogonal problem, and we simply assume that a stabilization after multi-component failures eventually allows reactive fault tolerance, such as restarts from a checkpoint, to occur in a synchronous manner. Sastry et al. discuss the impact of celerating environments due to heterogeneous systems where absolute speeds (execution progress) could increase or decrease [42]. Bichronal timers with the composition of action clocks and real-time clocks are able to cope with celerating environments. Our work, in contrast, only relies on the clock of a local node. Stephane et al. implemented a fault detector in a P2P-MPI environment utilizing a heartbeat approach [20]. They address failure information sharing, reason about a consensus phase and acknowledge overheads of fault detection due to their periodic heartbeat approach. Our work, in contrast, results in much lower message and time complexity. Consensus is orthogonal to our work, as discussed before.

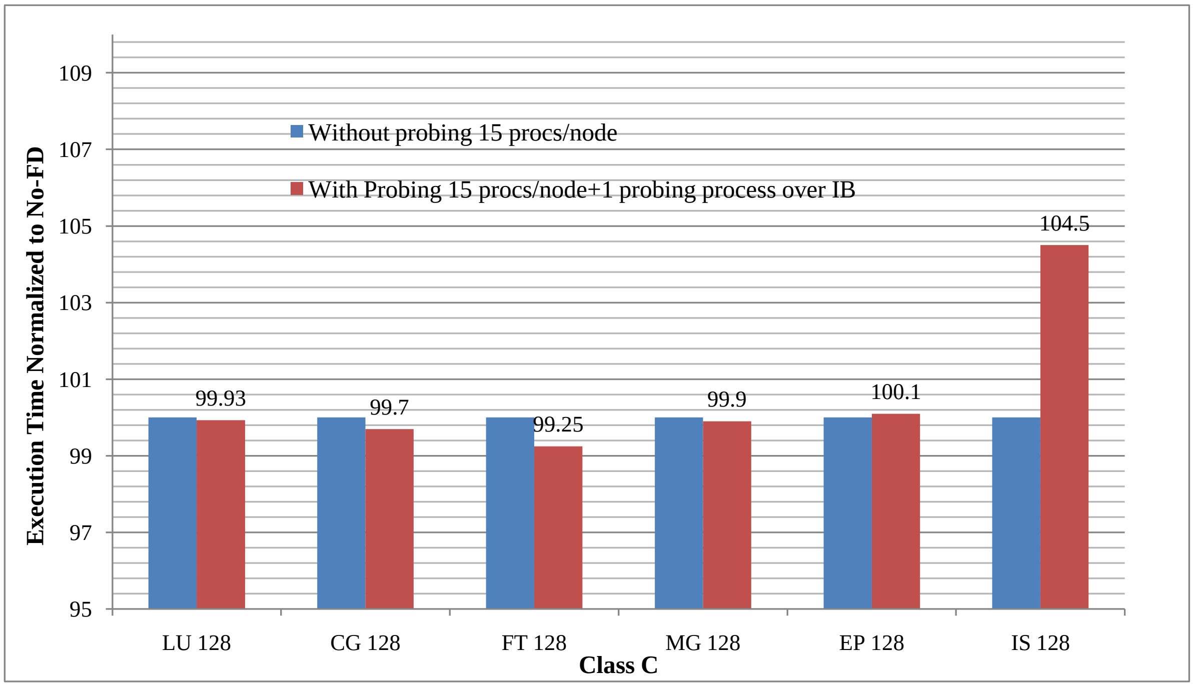


Figure 2.7: Background Probing 15 MPI Processes per Node

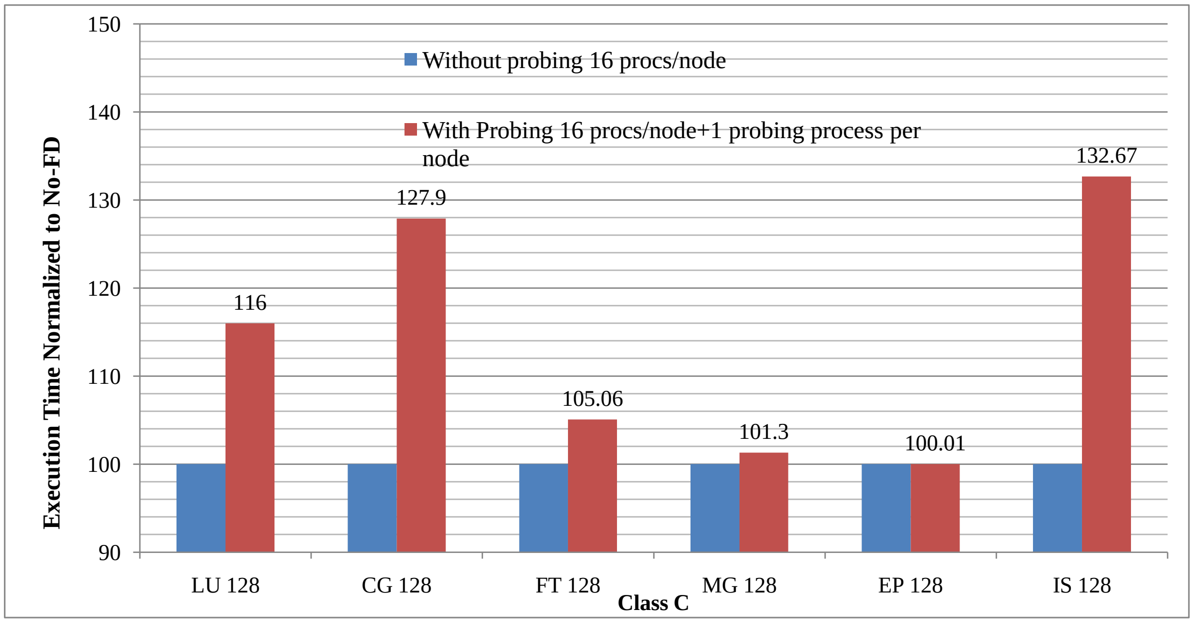


Figure 2.8: Background Probing 16 MPI Processes per Node

2.7 Conclusion

In summary, our work contributes generic capabilities for fault detection / liveness monitoring of nodes and network links both at the MPI level and stand alone. We designed and implemented two approaches to this end. The first approach utilizes a periodic liveness test and utilizes a ring-based network overlay for control messages. The second method promotes sporadic checks upon communication activities and relies on point-to-point control messages along the same communication paths utilized by the application, yet falls back to the ring-based overlay for collectives. We provide a generic interposing of MPI applications to realize fault detection for both cases plus a stand-alone version for the periodic case. Experimental results indicate that while the sporadic fault detector saves on network bandwidth by generating probes only when an MPI call is made, its messages are increasingly contending with application messages as the number of nodes increases. In contrast, periodic fault detection statistically avoids network contention as the number of processors increases. Overall, the results show that periodic failure detection performs better than sporadic for communication intensive codes and that separation of the FD from MPI applications reduces their perturbation. Failure detection can thus be implemented using Periodic probing that imposes an overhead of O(n) messages and O(1) time as opposed to $\Theta(n^2)$ and $\Omega(n)$ in the case of all-to-all probing and $\Theta(n)$ and $\Omega(log(n))$ in the case of tree-based liveness probing. Experiments show that periodic probing incurs an average overhead of 10% when integrated with the MPI application and a negligible overhead when failure detection runs independently in the background. This proves point one of the hypothesis stated in Section 1.4.

Chapter 3

Checkpoint/Restart with Redundancy

3.1 Checkpoint Restart

The idea behind the C/R technique is to capture the state of the application during failure-free operation and storing it to a stable storage. Upon failure of one or more processes, the application is restarted from the last successful checkpoint. A stable storage is an abstraction for a storage ensuring that recovery data persists through failures.

The C/R service provided by the MPI environment utilizes a single-process checkpoint service specified by the user as a plug-in facility. Depending upon the transparency with regard to 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 [44]. This requires a change in the application code to identify regions of memory that need to be part of the captured process image. *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, thus being able to capture the state of an entire process without being tied to a particular kernel [32]. *System level* checkpointing services are either implemented inside the kernel or as a kernel module [13]. The checkpoint images in system level checkpointing are not portable across different kernels. We have used Berkeley Lab Checkpoint Restart (BLCR) [13], which is a system level checkpoint service implemented as a kernel module for Linux, for performing our experiments.

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 protocols work before the individual process checkpoints are performed. A distributed snapshot algorithm [9], 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 currently implemented in Open-MPI [25] is an all-to-all bookmark exchange protocol, where processes exchange message totals between all peers and wait till 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, checkpoint/restart techniques come at an additional cost since performing checkpoints interrupts the normal execution of the application processes. There is also an overhead involved because of sharing of processors, I/O storage, network resources, etc. When assessing the cost of checkpoint/restart 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 [37, 46]. Checkpoint latency is the time required to create and establish a checkpoint to a stable storage. If the application is suspended until the checkpoint is established then the checkpoint overhead is equal to the checkpoint latency. Various optimization techniques have been studied to improve both forms of overhead, a short description of which follows.

Forked checkpointing forks a child process before the checkpoint operation is performed [10, 44]. The image of the child process is taken, while the parent process resumes execution. Afterword, the pages that have changed since fork are captured from the parent process. Thus, the task of performing and writing checkpoints to stable storage and execution of the application task are overlapped in time, thereby reducing the checkpoint overhead.

Checkpoint Compression is a method for reducing the checkpoint latency by reducing the size of process images before writing them to stable storage [44, 31]. *Memory Exclusion* skips temporary or unused buffers to improve checkpoint latency [38]. This is done by providing an interface to the application to specify regions of memory that can be safely excluded from the checkpoint [4]. *Incremental checkpointing* reduces the checkpoint latency by saving only the changes made by the application from the last checkpoint. These techniques commonly rely on the paging system of modern operating systems, like the modified or dirty bit in paging hardware of the MMU [21, 23]. During recovery, the incremental checkpoints are combined with the last full checkpoint to create a complete image of the process.

3.2 Redundancy Background

To decrease the failure rate for large-scale applications, redundancy can be employed at the process level [19, 29]. 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 which is considered to fail if all the nodes in the sphere become non-functional. Thus, the overall effect of using redundancy is that it increases the mean time between failures (MTBF) which in turn allows us to checkpoint less frequently for the same resiliency level.

RMPI [19] developed at Sandia National Laboratories is a user-level library that allows MPI applications to transparently use redundant processes. MR-MPI [15] is a modulo-redundant MPI solution for transparently executing HPC applications in redundant fashion which uses the PMPI layer of MPI. VolpexMPI [29] is a MPI library implemented from scratch that supports redundancy internally with an 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 developed at the System Research Lab at NCSU in collaboration with Oak Ridge National Laboratories and Sandia National Laboratories, which 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 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 and simulations.

3.2.1 Design of Redundancy

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 source code of the application. The redundancy module is activated when a call is made to MPI_Init(), at which time it 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 an MPI messages to/from all the replicas of the receiver/sender process.

Consider the scenario shown in Figure 3.1, 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 process.

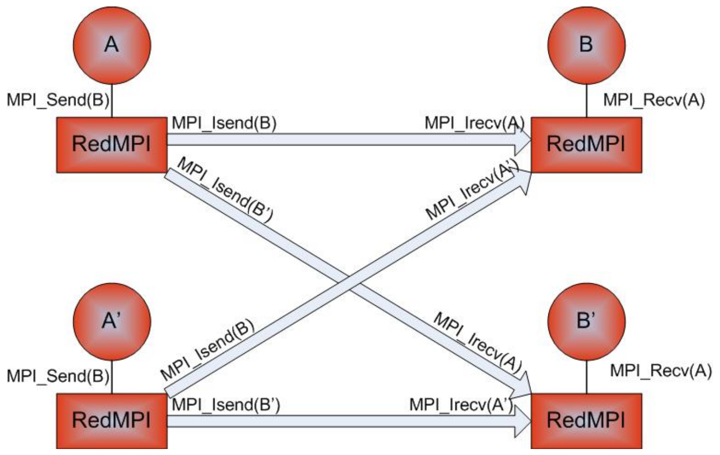


Figure 3.1: Redundancy: Blocking Point to Point Communication

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 the redundant partners of the sender processes that are alive. The RedMPI library allocates additional buffers for receiving redundant copies of the messages. 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 3.2 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 when MPI_ANY_SOURCE is used in a MPI_Recv() operation. Since a message sent from any process can complete a MPI_ANY_SOURCE receive request, we have to make sure that all the replicas of the process get message from the same sender. To ensure this, RedMPI performs the following steps:

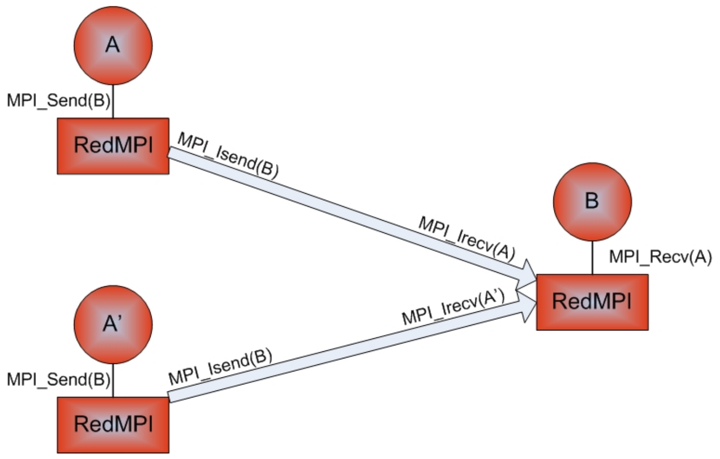


Figure 3.2: Partial Redundancy: Blocking Point to Point Communication

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 get the message from the remaining replicas of the sender process determined above.
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 node 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 handlers returned by all the non-blocking MPI calls. A request handler that acts as an identifier to this set is returned to the user application. When the application, at a later point, makes a call to MPI_Wait(), RedMPI waits on all the requests belonging to the set before returning from the call.

3.3 Motivation

Checkpoint/Restart is the most commonly used method to recover from faults while running HPC applications. But this technique suffers from scalability limitations. As we increase the number of cores, the amount of useful work performed decays rapidly. This can be verified by a study from Sandia National Lab [19], data from which is shown in Table 3.1. Only 35% of the total work done is due to computation for a 128 hour job running on 100k nodes with a MTBF of 5 years.

Table 3.1: Checkpoint/Restart Overhead for an Application Performing 128 hours of Work, with MTBF 5 years

No of Nodes	Work	Checkpoint	Recomputation	Restart
100	96%	1%	3%	0%
1,000	92%	7%	1%	0%
10,000	75%	15%	6%	4%
100,000	35%	20%	10%	35%

We can employ redundancy to cut down the failure rate of the MPI application, which would result in less overhead for performing checkpoints and repeated execution. The downside of redundancy is that it requires additional computing elements depending on the degree of redundancy. There is also an increase in the total execution time due to redundant communication.

In this thesis, we try to answer following questions:

- Is it advantageous to use both the techniques at the same time to improve performance or job throughput?
- What are the optimal values for the 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 user may want to execute their application with least number of required resources. The user may also create a cost function giving different weights to execution time and number of resources used.

In this thesis, we create 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.

3.4 Assumptions

We make the following assumptions in our model about the execution environment.

- We assume that spare resources are used for performing redundant computation. This means that if an application using 'N' nodes moves to 2x redundancy, it now uses '2N' nodes for computation. This assumption gaurantees that redundancy does not slow down the computation of the application.

- We assume Poisson arrivals of node failures. The interval between failures in this case is given by an exponential distribution.
- The failure model we assume here is fail-stop failures. Fail-stop failures are the most frequent ones and easier to detect than other types of non-crash failures.

3.5 Mathematical Analysis

3.5.1 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 of replica processes which perform the same task and are hidden from each other at the application level is called a virtual process. The processes inside a sphere seen by the underlying system are called as 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 which 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:

- number of virtual processes involved in the parallel application = N,
- redundancy degree (number of redundant processes per virtual process) = r,
- total number of physical processes = $N \cdot r$,
- base execution time of the application = t,
- mean Time to Failure (MTBF) of each node = θ.

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 and the degree of replication.

In the analysis developed here, we consider overhead due to redundant communication but ignore overheads caused by other factors.

Let α be the communication/computation ratio of 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, where r is the redundancy level (typically 2 or 3).

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

$$t_{Red} = (1 - \alpha)t + \alpha t r \tag{3.1}$$

2. We assume that the arrival of failures for each component is Poisson process. Hence, reliability of a physical process, which is the probability that the process survives for time t is:

$$R(t) = e^{-t/\theta} \tag{3.2}$$

When θ is large, it can be approximated as:

$$R(t) = 1 - t/\theta \tag{3.3}$$

Hence, the probability that a node survives for the entire duration t_{Red} of application execution is:

$$P(Survival) = 1 - t_{Red}/\theta$$

Using the above result, the probability of failure of a node over the time period t_{Red} is:

$$P(Failure) = 1 - (1 - t_{Red}/\theta) = t_{Red}/\theta \tag{3.4}$$

3. Each virtual process has r physical processes, implying the following relation:

   Reliability of a virtual process

   = probability that at least one physical process survives

   = 1 - (Probability that all physical processes fail)

   $= 1 - (t_{Red}/\theta)^r$

4. Since failure of a 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} = Probability that all the virtual nodes survive until the end

$$R_{sys} = [1 - (t_{Red}/\theta)^r]^N \tag{3.5}$$

5. Equation 3.2 can be written in terms of failure rate (λ) as:

$$R(t) = e^{-\lambda t}$$

   Using above equation, the failure rate of the system can be obtained as,

$$\lambda_{sys} = -log(R_{sys})/t_{Red}$$

$$\lambda_{sys} = -N\frac{log[1 - (\frac{t_{Red}}{\theta})^r]}{t_{Red}} \tag{3.6}$$

Figure 3.3 shows how varying r (the degree of redundancy) changes the reliability of the entire system as a function of r, given the indicated sample input parameters. Studies [43] have shown that failure of a node depends on the number of sockets and not on the number of cores comprising that node. Researchers usually consider a socket as a unit of failure and refer to the number of sockets when measuring system reliability. The MTBF/node of 5 years mentioned in this Figure actually refers to MTBF of each socket. If the node is a dual-socket node then its MTBF is, in fact, 10 years, which is consistent with [43]. But for simplicity we refer to nodes in this document.

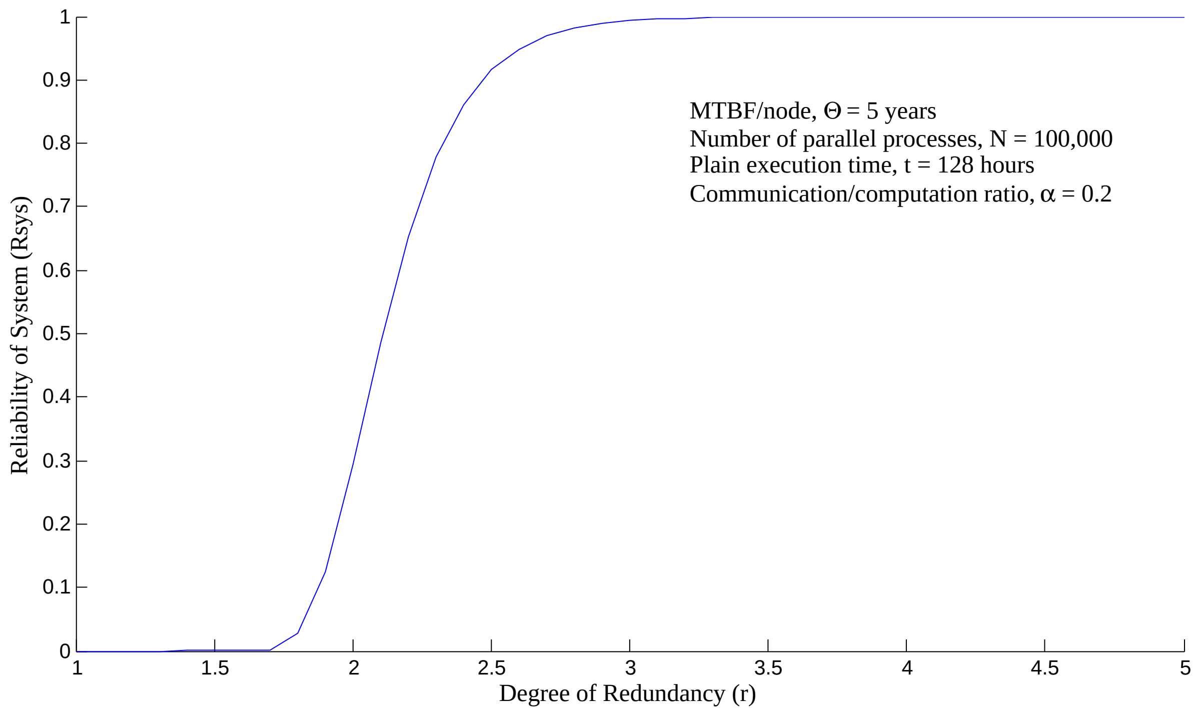


Figure 3.3: Effect of Redundancy on Reliability

3.5.2 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 regular time intervals.

As discussed earlier, performing checkpoints comes at a cost. Each checkpoint taken of a parallel application has certain overheads depending on various parameters including number of parallel tasks, time taken to synchronize the processes and time taken to store checkpoints to stable storage.

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

Consider a parallel application with the following parameters,

- t : Time taken by the original application to complete in absence of failure.
- λ : 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. It 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. If spare nodes are not readily available, then the time required for repair/replacement is also included.

Figure 3.4 shows the lifetime of a process in the presence of periodic checkpointing and occurrence of failures.

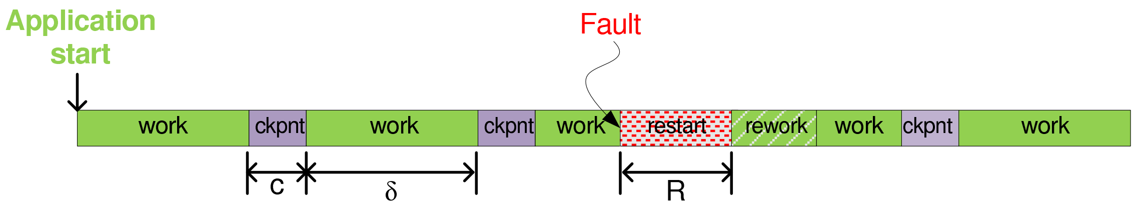


Figure 3.4: Life-cycle of an application

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 rework time, i.e., the time taken for recomputing the lost work between the last checkpoint and the occurrence of failure. Again, with the assumption that failures occur only during the computation phase (no failures occur during the rework phase), the average amount of rework is half the time between successive checkpoints = $\delta/2$.

   Thus, the rework time during the entire run of the application is:

$$\text{(Number of failures)} \times \text{(Average rework time)} = (t \times \lambda) \times (\delta/2)$$

4. The total restart overhead derived from the total number of failures during the lifetime

of the application:

$$(\text{Number of failures}) \times (\text{Restart overhead per failure})$$

$$= (\text{Total application execution time} \times \text{Failure rate}) \times R$$

$$= t\lambda R$$

Thus,

$$\text{The total time is: } T = t + \frac{tc}{\delta} + t\lambda\frac{\delta}{2} + t\lambda R \tag{3.7}$$

Our goal is to choose δ such that the total time required is minimal. To find the minimal value, we construct the first order derivative of above equation:

$$\frac{dT}{d\delta} = \frac{d}{d\delta}(t + \frac{tc}{\delta} + t\lambda\frac{\delta}{2} + t\lambda R)$$

$$\frac{dT}{d\delta} = \frac{-tc}{\delta^2} + t\lambda\frac{1}{2}$$

The second order derivative is:

$$\frac{d^2T}{d\delta^2} = \frac{2tc}{\delta^3}$$

Solving the equation $\frac{dT}{d\delta} = 0$, we get:

$$0 = \frac{-tc}{\delta^2} + t\lambda\frac{1}{2}$$

$$\delta = \sqrt{\frac{2c}{\lambda}} \tag{3.8}$$

The value of the second order derivative at $\delta = \sqrt{\frac{2c}{\lambda}}$ is:

$$\frac{d^2T}{d\delta^2} = \frac{2tc}{(\frac{2c}{\lambda})^{\frac{3}{2}}} > 0 \qquad \text{since c, t, and } \lambda \text{ are} > 0$$

This result shows that to complete the task in the least time (i.e., with the least overhead), the checkpointing interval should be chosen as $\sqrt{\frac{2c}{\lambda}}$.

Thus, the optimal checkpointing interval can be expressed in terms of the mean time between failures (θ) as $\sqrt{2c\theta}$.

The minimum is obtained by substituting δ in 3.7:

$$T_{min} = t + \frac{tc}{\sqrt{\frac{2c}{\lambda}}} + t\lambda\frac{\sqrt{\frac{2c}{\lambda}}}{2} + t\lambda R$$

$$= t + t\sqrt{2c\lambda} + t\lambda R$$

In summary, we get:

$$T_{min} = t(1 + \sqrt{2c\lambda} + \lambda R) \tag{3.9}$$

3.5.3 Combining Redundancy and Checkpointing

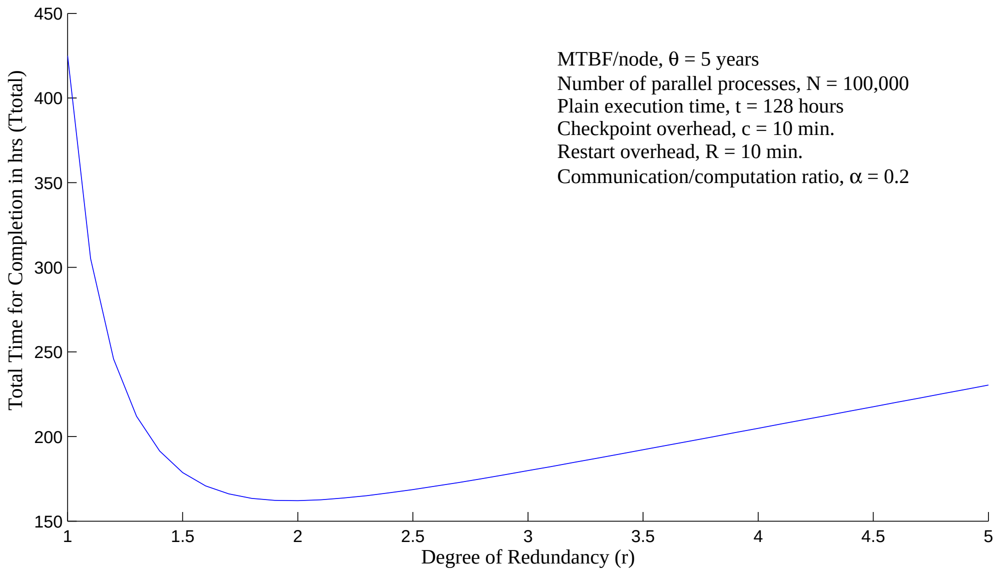


Figure 3.5: Configuration 1: Total Execution Time with Varying Degree of Redundancy

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.

Thus, we need to checkpoint the application so that we can recover the application after a failure instead of running the application from the scratch. Equation 3.9 shows that the least possible time required to complete the application increases as the failure rate increases (equivalently, the MTBF decreases).

From Equation 3.6, we can 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 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.

- The application time with redundancy degree r is:

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

- The failure rate with redundancy degree r is:

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

- The total time required for execution is:

$$T_{total} = t + t\sqrt{2c\lambda_{sys}} + t\lambda_{sys}R$$

Figure 3.5 shows the variation in total time with varying degree of redundancy of an application for an original running time of 128 hrs takes. We see that as we increase the degree of redundancy, the execution time decreases initially. The minimum time achieved is at 165 hours when the redundancy degree is ≈ 2. As we increase the redundancy further, the total time increases as well.

Figure 3.6 shows the execution time with the same configuration as above, but the MTBF of a node is increased to 10 years. As seen in the graph, the minimum execution time is obtained at a lower redundancy degree of ≈ 1.5.

Figure 3.7 shows the variation in execution time when the single checkpoint overhead is 1 minute compared to 10 minute in configuration 1. The effect of this is that the minimum execution time is obtained at a lower redundancy degree of ≈ 1.5 as compared to configuration 1.

3.6 Simulation framework

Many assumptions and approximations had to be made while performing the mathematical analysis. The most significant one is in Equation 3.1 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 be different from that shown in Figures 3.5, 3.6 and 3.7 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

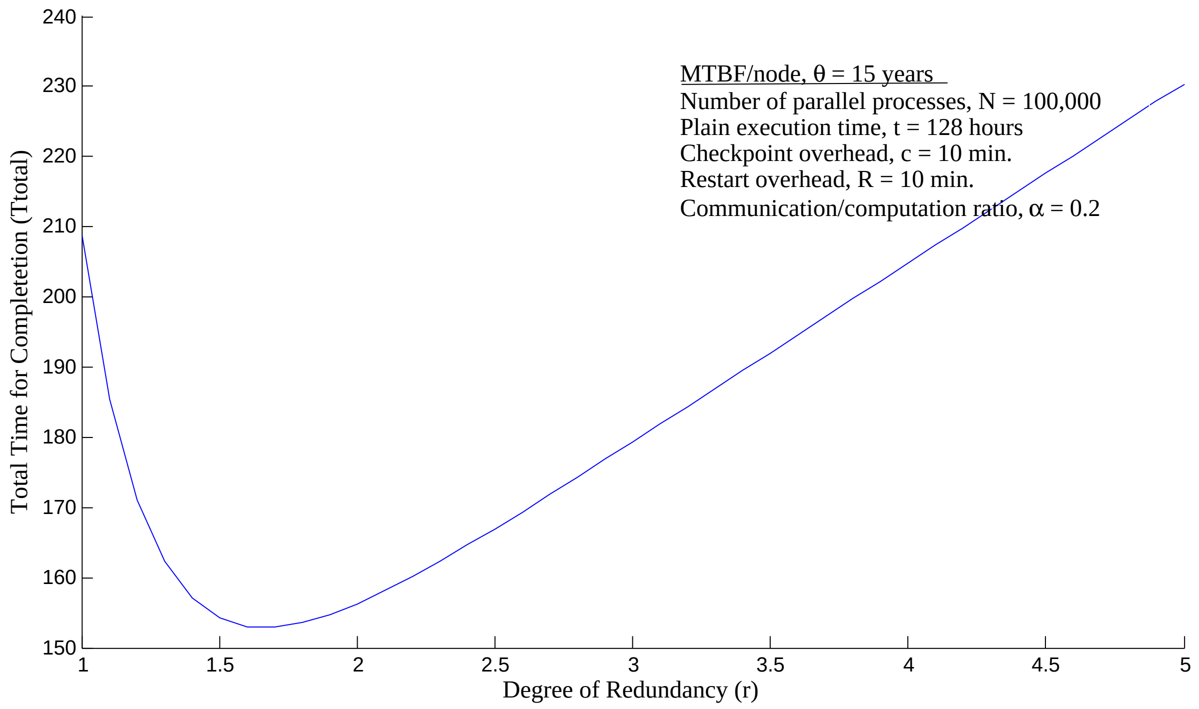


Figure 3.6: Configuration 2: Total Execution Time with Varying Degree of Redundancy

current available resources. We run the application with a certain degree of redundancy and also checkpoint the application at the optimum frequency calculated from Equation 3.8. Two processes run in the background of the application, which perform the two tasks specified above.

The first background process is the failure simulator that simulates failures of the entire application by simulating per-process failures. The occurrence of a failure for each process is assumed to be a Poisson process.

The simulator 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, time for next failure is calculated using an exponential distribution that describes the time between events in 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.

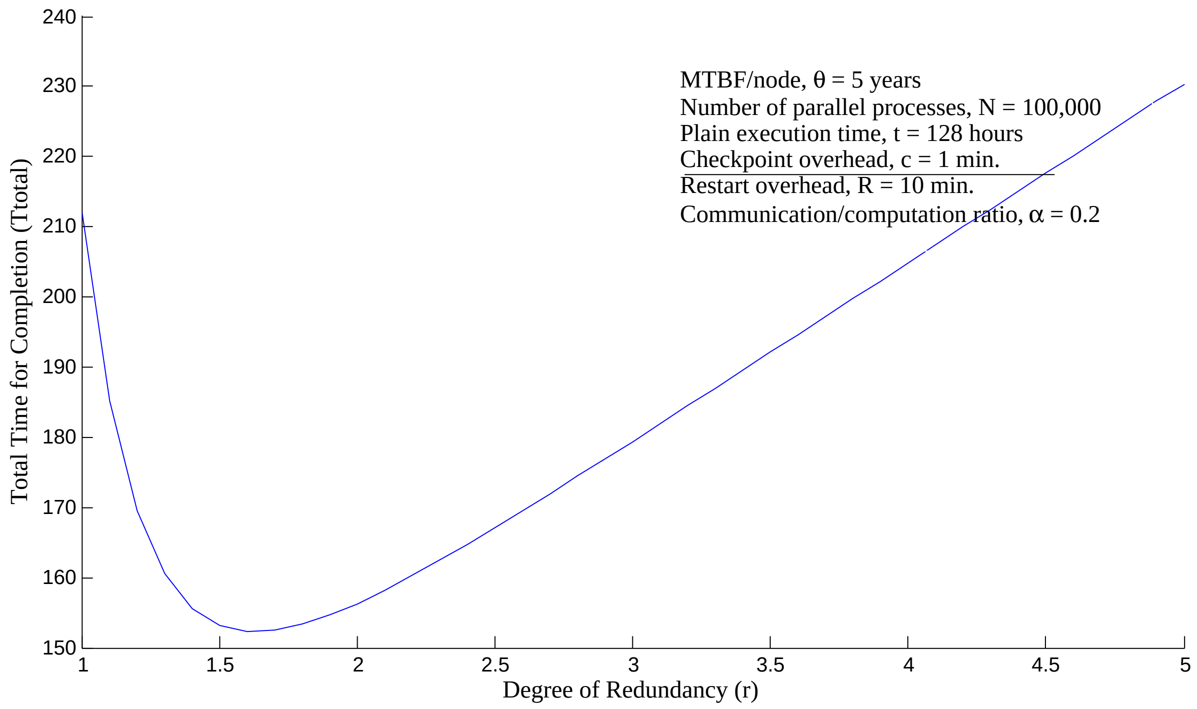


Figure 3.7: Configuration 3: Total Execution Time with Varying Degree of Redundancy

Figure 3.8 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 3.8 and 3.6 . It sets a timer for time δ and checkpoints the application when the timer goes off. The checkpoint is stored to stable storage and a pointer to the latest checkpoint image is maintained in the failure simulator process.

3.7 Results

Experimental platform: Experiments were conducted on a 108 node cluster with QDR Infiniband. Each node is a 2-way 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 simulations 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 for simulations. 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

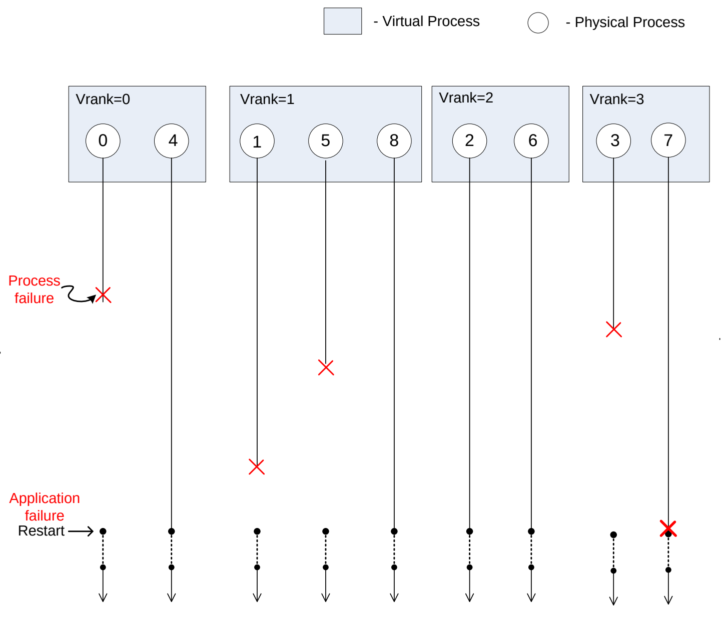


Figure 3.8: Simulating Failure for a MPI Application

typical of unstructured grid computations in that it tests irregular long distance communication employing unstructured matrix vector multiplication. CG takes approximately 20% of the total time for MPI communication. Thus, it is ideal for running our simulations. 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 compiled for 128 processes takes 46 minutes under failure free execution without redundancy and CR.

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 simulator 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. Simulations 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 only 50% of the processes have replicas.

Partial redundancy was employed in steps of 0.25x between 1x and 3x.

The results of the simulation for the optimal application execution time achieved using various degrees of redundancy is shown in Table 3.2. 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 MTBF of 18, 24 and 30 hrs, the maximum performance is achieved at 2x redundancy. Figures 3.9 and 3.10 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.

Table 3.2: Performance of an Application (Execution Time in Minutes) with Combined CR+Redundancy Technique

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

We make the following observation from the above results:

- 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).
- 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.
- 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.
- An interesting finding is that in most cases 1.25x redundancy yields poor performance as compared to 1x (when no redundancy is employed). Similarly, 2.25x yields poor perfor-

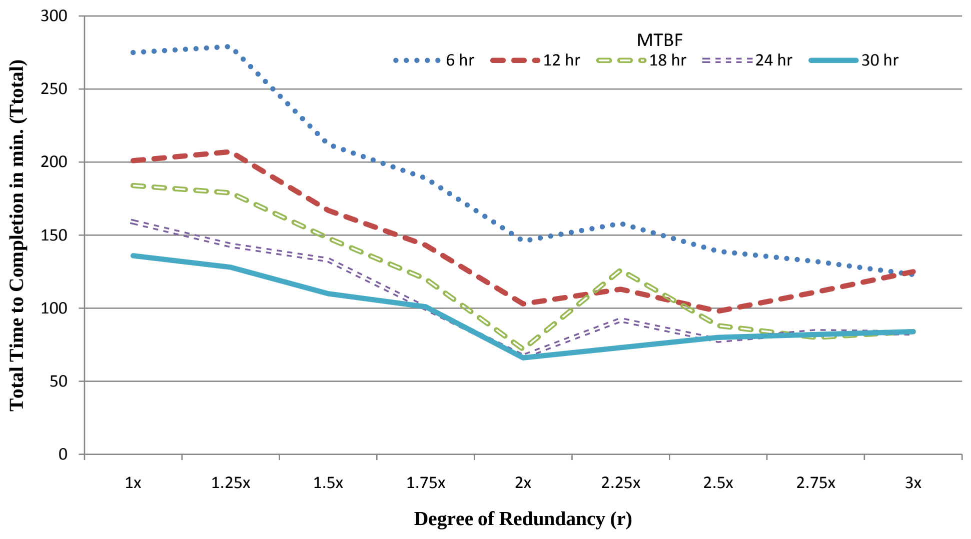


Figure 3.9: Simulating Results - Performance of an Application (Execution Time in Minutes) with Combined CR+Redundancy Technique

mance 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 3.3 and Figure 3.11. 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.

Table 3.3: Increase in Execution Time with Redundancy

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

- The purpose of running these simulations is to verify the mathematical model developed in Section 3.5. Hence, we modeled our execution by estimating/calculating the environment parameters and substituting them in the set of equations developed in Section 3.5.3. The overhead per checkpoint (c) was calculated as 120 sec. by first running the plain ap-

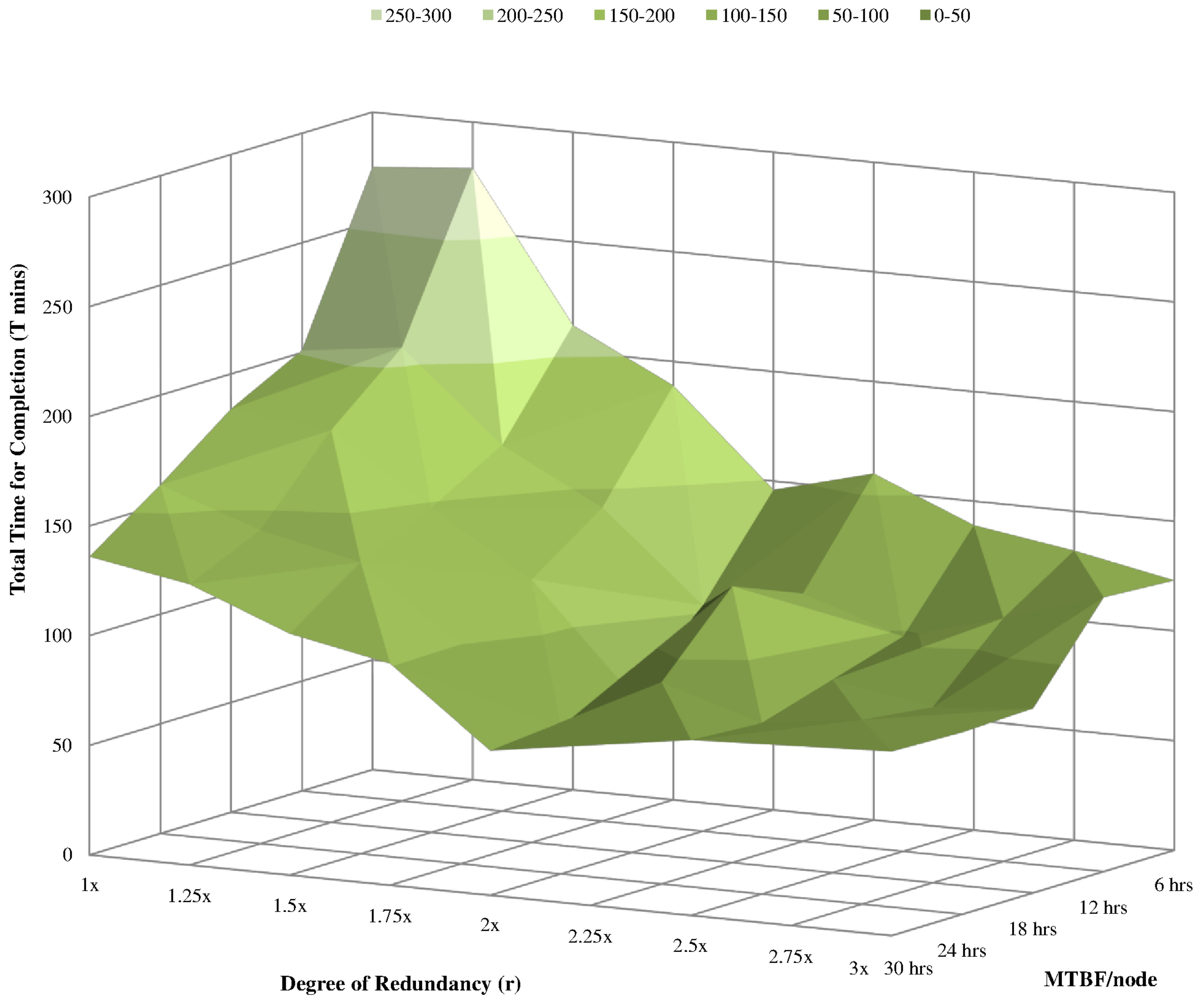


Figure 3.10: Surface Plot of Simulating Results - Performance of an Application (Execution Time in Minutes) with Combined CR+Redundancy Technique

plication, 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

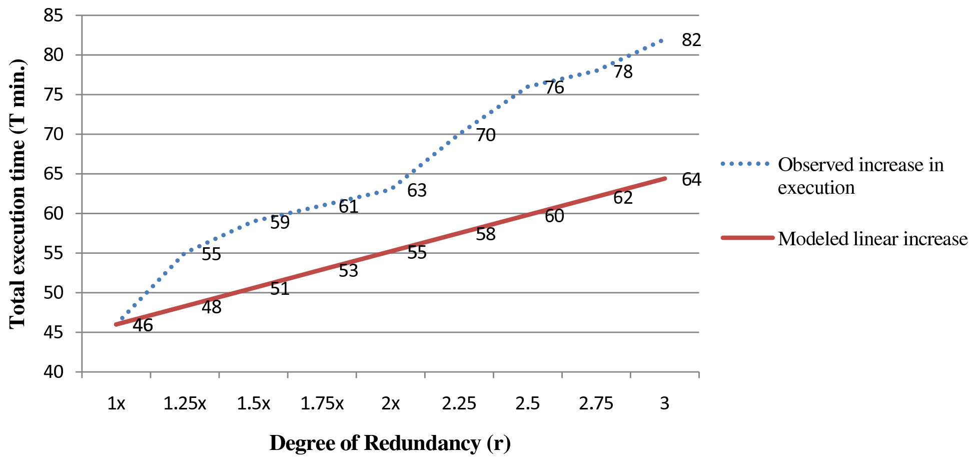


Figure 3.11: Increase in Execution Time with Redundancy

the equations in MATLAB, we get the expected application behavior shown in Figure 3.12. It can be seen that the actual behavior of the application (Figure 3.9) is similar to the modeled behavior shown in Figure 3.12, thus validating our analytical model. For closer comparison, Figure 3.13 overlays the performance curves in Figure 3.12 over those in Figure 3.9 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 sources listed below :

- The redundancy overhead in actual runs is more than the modeled overhead as demonstrated in Figure 3.11. An increase in the execution time is due to additional failures occurring during this extra time.
- The model uses average rework time, which is half of the interval between two checkpoints (Section 3.5.2). In real runs, the occurrence of failures may deviate from the assumed mid-checkpoint occurrence.
- There is some randomness involved while running the simulations because of the fact that the model uses the average failure rate (λ) while the simulations generate Poisson failures from a random number generator.

3.8 Related Work

Several models to determine the optimal checkpointing strategy for parallel programs have been developed in prior works. Young [52] presented an optimal checkpoint and recovery model and obtained a constant optimal checkpoint interval to reduce the overall execution time. Based on

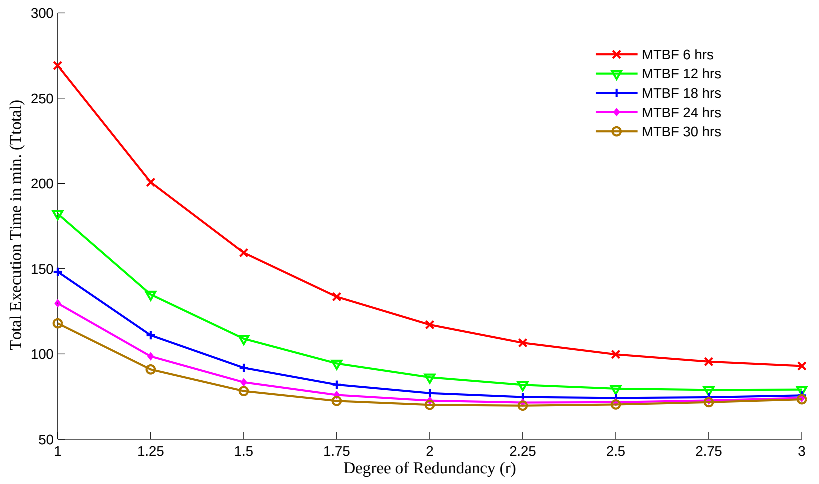


Figure 3.12: Modeled Application Performance

Youngs work, Daly [11] 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 3.5.2.

Authors of [38], [50] have taken a different approach by modeling the problem as a Markov availability model and obtained an optimal checkpoint placement that maximizes system availability. [38] 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 [33], authors have presented a reliability-aware method for an optimal checkpoint/restart 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 checkpoint/restart 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. [18] have studied the viability of process replication as the primary fault tolerance mechanism for exascale systems, employing checkpoint/restart as a secondary mechanism. Results from their work show that replication outperforms traditional checkpoint/restart approaches for large sockets counts

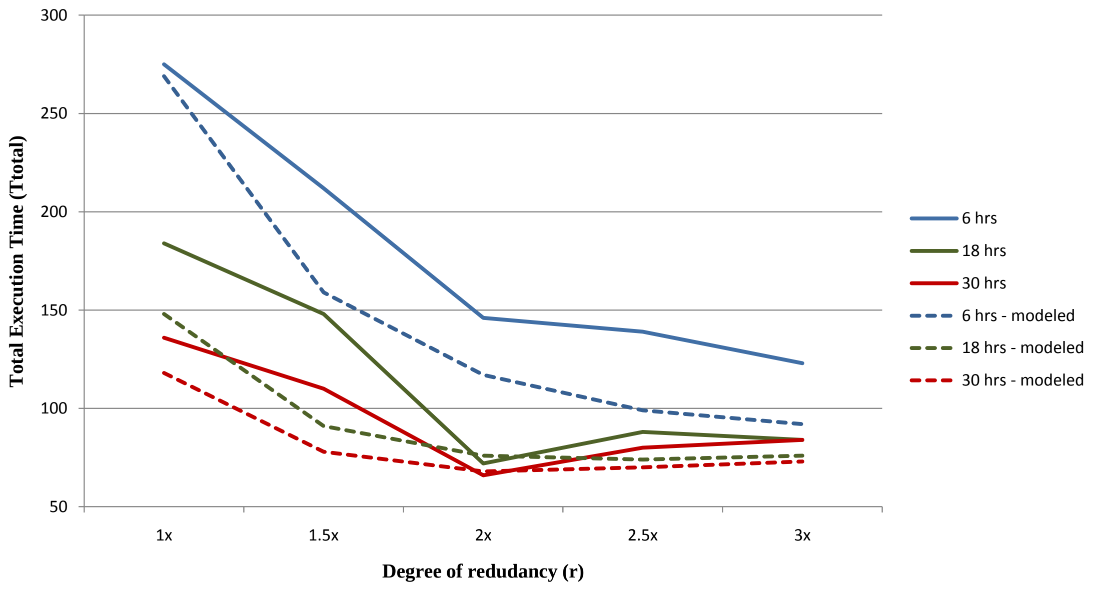


Figure 3.13: Observed Performance Overlayed on Modeled Performance

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. In this thesis, we model the execution of an application in the presence of redundancy at various degrees (including partial redundancy) in combination with checkpoint/restart. Using this model, we study the trade-off between levels of redundancy and checkpoint frequencies with the goal of optimizing system performance.

3.9 Conclusion and Future Work

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 checkpoint/restart (CR) activities. Another way to achieve fault tolerance is to employ redundancy, wherein multiple processes perform the same computation. This work shows that CR-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 a long-running large-scale programs in presence of failures using CR-based and redundant-based fault tolerance. 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 simulating an exascale environment on our computing cluster. The modeled application behavior closely mimics the observed application behavior 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, the observed times do not follow exactly the same curve especially at partial redundancies. The reason for such behavior was traced to the deviation of observed redundancy overhead from the expected overhead. Thus, we have proved point two of the hypothesis stated in Section 1.4.

This work can be extended by improving the models of application execution time under redundancy and that of overhead of C/R fault tolerance. Following are the additions that can be made to the model :

- The analysis of Section 3.5.1 relating the increase in execution time to redundancy does not exactly follow the observed increase in execution time (Figure 3.11). This analysis can be improved by studying the indirect effects of partial redundancy. For example, when some process performs redundant communication, it may slow down other processes that do not perform redundant communication.

- Furthermore, different schemes for placement of redundant processes and their effect on the failure rate and application execution time can be studied.

- With regards to CR-based fault tolerance, we can extend the model by incorporating advanced C/R techniques, such as incremental checkpointing [44, 31], into the analysis.

- We can also consider optimizing the overhead by employing variable checkpointing frequency as done in related studies.

- In addition, this study neglects the increase in checkpointing overhead when redundant processes take part in execution. Most of the current checkpointing schemes store process images to local disk, which are then written back to a centralized storage. So there is a negligible effect of redundancy on checkpoint overhead, given that the number of processes per node remains constant. The model can be enhanced to study and incorporate the effect of redundancy on checkpoint overheads.

Chapter 4

Summary

With the rapid increase in the number of cores and nodes in high performance computing (HPC), exascale computing will soon become a reality. Harnessing such computing power is a challenging task as system reliability decays rapidly with increasing number of cores. To make applications run longer than the mean time between failures (MTBF) various methods have been proposed such as transparent checkpointing, process replication, message logging, etc. Each of the approaches require the underlying system to have the capability to detection process faults. First part of the thesis covered in Chapter 2 addressed the issue of detecting failures of processes participating in a parallel MPI task. We studied and evaluated two kinds of failure detection mechanisms. From the experiments we conclude that periodic failure detection performs better than sporadic for communication intensive codes. Performing periodic failure detection outside of the application gives negligible overhead.

Chapter 3 describes the second part of the thesis work. Current practices in HPC community shows that checkpoint/restart (CR) is the most common fault tolerance technique used for HPC. However exascale applications are projected to spend majority of their time performing CR related activities, giving less than 50% utilization. Another fault tolerance mechanism is to use replication to mask process failures. Apart from the additional resources required, this scheme comes with a execution overhead. Second half of the work analyses the combined effect of the two approaches on application behavior. We developed a theoretical model representing the execution of an application in a failure-prone environment using the combined scheme. This model gives us the execution time of the application for given degree of redundancy and user-specified parameters like per-checkpoint overhead, restart time, etc. Using this model we obtain the degree of redundancy which gives us the maximum performance. We verified our model by running simulations on a real HPC-like environment. The model can be used by HPC application users to determine the best configuration of redundancy degree and checkpoint frequency.

REFERENCES

[1] Message passing interface MPI. https://computing.llnl.gov/tutorials/mpi.

[2] Top 500 list. http://www.top500.org/, June 2002.

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

[4] 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.

[5] A. Agbaria and R. Friedman. Starfish: Fault-tolerant dynamic mpi programs on clusters of workstations. In *Proc. of the 8th IEEE Intl. Symp. on High Perf. Distr. Comp.*, 1999.

[6] D. H. Bailey et al. The NAS Parallel Benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[7] G. Bosilca, A. Boutellier, and F. Cappello. MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes. In *Supercomputing*, November 2002.

[8] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, March 1996.

[9] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions Computer Systems*, 3.

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

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

[12] J. Duell. The design and implementation of berkeley lab's linux checkpoint/restart. Tr, Lawrence Berkeley National Laboratory, 2000.

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

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

[15] Christian Engelmann and Swen Böhm. Redundant execution of HPC applications with MR-MPI. In *Proceedings of the* 10th *IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN) 2011.*

[16] Graham E. Fagg and Jack J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. In *Euro PVM/MPI Meeting*, volume 1908, pages 346–353, 2000.

[17] 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.

[18] 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.

[19] 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.

[20] S. Genaud and C. Rattanapoka. Evaluation of replication and fault detection in p2p-mpi. In *Intl. Par. and Distrib. Proc. Symp.*, 2009.

[21] 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.

[22] Junyoung Heo, Sangho Yi, Yookun Cho, Jiman Hong, and Sung Y. Shin. Space-efficient page-level incremental checkpointing. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 1558–1562, New York, NY, USA, 2005. ACM.

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

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

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

[26] Joshua Hursey, Jeffrey M. Squyres, Timothy I. Mattox, and Andrew Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *12th IEEE Workshop on Dependable Parallel, Distributed and Network-Centric Systems*, 03 2007.

[27] Failure Detection in MPI programs Kamal KC.

[28] Hideyuki Jitsumoto, Toshio Endo, and Satoshi Matsuoka. Abaris: An adaptable fault detection/recovery component framework for mpis. In *Intl. Par. and Distrib. Proc. Symp.*, 2007.

[29] 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.

[30] 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.

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

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

[33] 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*, 2007.

[34] John Mehnert-Spahn, Eugen Feller, and Michael Schoettner. Incremental checkpointing for grids. In *Linux Symposium*, July 2009.

[35] A. Moody, G. Bronevetsky, Kathryn Mohror, and Bronis de Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Supercomputing*, November 2010.

[36] 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.

[37] 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.

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

[39] 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.

[40] J.F. Ruscio, M.A. Heffner, and Srinidhi Varadarajan. Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Intl. Par. and Distrib. Proc. Symp.*, 2007.

[41] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, Andrew Lumsdaine, Jason Duell, Paul Hargrove, and Eric Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *Proceedings, LACSI Symposium*, October 2003.

[42] Srikanth Sastry, Scott M. Pike, and Jennifer L. Welch. Crash fault detection in celerating environments. In *Intl. Par. and Distrib. Proc. Symp.*, 2009.

[43] B. Schroeder and G. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN-2006)*, Philadelphia, PA, June 2006.

[44] 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.

[45] G. Stellner. CoCheck: checkpointing and process migration for MPI. In IEEE, editor, *Proceedings of IPPS '96. The 10th International Parallel Processing Symposium: Honolulu, HI, USA, 15–19 April 1996*, pages 526–531, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1996. IEEE Computer Society Press.

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

[47] J. Varma, C. Wang, F. Mueller, C. Engelmann, and S. L. Scott. Scalable, fault-tolerant membership for MPI tasks on hpc systems. In *International Conference on Supercomputing*, pages 219–228, June 2006.

[48] C. Wang, F. Mueller, C. Engelmann, and S. Scott. A job pause service under LAM/MPI+BLCR for transparent fault tolerance. In *Intl. Par. and Distrib. Proc. Symp.*, April 2007.

[49] C. Wang, F. Mueller, C. Engelmann, and S. Scott. Proactive process-level live migration in hpc environments. In *Supercomputing*, 2008.

[50] K. Wong and M. Franklin. Distributed computing systems and checkpointing. In *High Performance Distributed Computing, 1993., Proceedings the 2nd International Symposium on*, 1993.

[51] Sangho Yi, Junyoung Heo, Yookun Cho, and Jiman Hong. Adaptive page-level incremental checkpointing based on expected recovery time. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1472–1476, New York, NY, USA, 2006. ACM.

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