Computational Resiliency:

Reliable Heterogeneous Applications

by

Joohan Lee

M.S. Sogang University, Korea. 1995

B.S. Sogang University, Korea. 1993

DISSERTATION

Submitted in partial fulfillment of the requirements for the

Degree of Doctor of Philosophy in Computer and Information Science

in the Graduate School of Syracuse University

October 2001

Advisor: Professor Steve J. Chapin

Abstract

This thesis presents the notion of *computational resiliency* to provide reliability in heterogeneous distributed applications. The notion provides both software fault tolerance and the ability to tolerate information warfare (IW) attacks. This technology seeks to strengthen a military mission, rather than protect its network infrastructure using static defense measures such as network security, intrusion sensors, and firewalls. Even if a failure or successful attack is never detected, it should be possible to continue information operations and achieve mission objectives.

Computational resiliency involves the dynamic use of replicated software structures, guided by mission policy, to achieve reliable operation. However, it goes further to automatically regenerate replication in response to a failure or attack, allowing the level of system reliability to be restored and maintained. Replicated structures can be protected through several techniques such as camouflage, dispersion, and layered security policy. This thesis examines a prototype concurrent programming technology to support computational resiliency in a heterogeneous distributed computing environment. The performance of the technology is explored through two example applications, concurrent sonar processing and remote sensing.

We develop the associated performance analytical model and verify the model against the experimental results. Overhead of computational resiliency over homogeneous and heterogeneous systems are investigated. Load balancing techniques are used to improve the overall performance of the system especially on heterogeneous computing environments.

Committee Approval Page

Table of Contents

List of Illustrative Materials

Figures

Chapter 1 Introduction

Any system that operates in highly adverse environments, such as battlefield command and control, must be able to operate reliably by tolerating failures and attacks. Many distributed systems have sought to use state replication, either in hardware or software, as a mechanism to provide fault-tolerance and recovery. These approaches provide graceful degradation of performance to the point where no further replicas are available and then system failure occurs. This is not sufficient to *assure* information operations in adverse military situations where networked resources may become available dynamically through retasking.

1.1 General Approach

We are investigating an alternative model of distributed computation termed *computational resiliency.* This model combines real-time attack assessment with process reconfiguration, dispersion, camouflage, on-the-fly replication, and layered security policy to reliably maintain information operations. To visualize how these concepts might operate, consider a distributed application as analogous to an apartment complex inhabited by a new strain of roach (a process or thread)[1]. The roaches are highly resilient: you can stamp on them, spray them, strike them with a broom but you never kill them all or prevent them from their goal of finding food (resources). To foil your eradication

1 Thanks to Cathy McCullum for providing this analogy.

efforts, they use several techniques: (1) they are *highly mobile* moving from one place in the apartment complex (network) to another with speed and agility. (2) they continually *replicate* to ensure that it is not possible to kill them all. (3) they *sense* their environment (attack assessment) to obtain clues that mobility is necessary; if a light is turned on, they scurry away in all directions to hide behind cupboards in places of *known safety* (secure network zones). (4) if a new roach killer is invented they *learn* from it, and *adapt* their behavior to compensate. However, this new strain is particularly aggressive and seeks to live in the daylight (wide-area operation); thus it adopts techniques for camouflage as a form of protection and disinformation.

To support this model, we have developed an application-independent programming technology that operates in heterogeneous distributed computing environments. The technology can be applied either to an entire application or a small number of selected components that are crucial to reliable operation. It incorporates the notion of resiliency into an application through a novel message-passing library. The library hides the details of the communication protocols required to achieve automatic on-the-fly replication and reconfiguration. It operates on a broad variety of networked architectures that include commercial-off-the-shelf computer systems and networking components, shared-memory multiprocessors and clusters of homogeneous machines. The library distinguishes these architectural differences for the purpose of performance improvement. For example, when communicating within shared memory, pointer copying is used; when communicating within a homogeneous cluster, no byte or machine translations are needed.

Since machines in the environment may have widely different performance and memory characteristics, load balancing techniques are required. These techniques must disperse replicated structures to realize improved reliability. To explore the performance issues associated with these concepts, we have incorporated the technology into two prototype distributed applications: a towed-array sonar and a hyper-spectral remote sensor. In this thesis we outline the applications, and show how resiliency is applied to them. Performance measurements are provided that quantify the overhead of resiliency, under normal operating conditions, using a network architecture of both homogeneous multi-processors and heterogeneous computers connected with both Gigabit and Fast Ethernet technologies.

Analytical models have been developed to understand the performance characteristics of the example applications with computational resiliency, and they can be used to predict the runtimes of the applications with respect to different reliability requirements.

1.2 Thesis Statement

THESIS

Scalable, transparent, and automated reconfiguration and recovery from failures and attacks can be achieved by strengthening applications using replicated structures in heterogeneous distributed environments.

Three principles were used to guide the development of the mechanisms described by the thesis statement. Each principle addresses part of the thesis statement, and together they form a basis for constructing resilient support mechanisms that fulfill the thesis.

Transparency The methods to provide computational resiliency should be transparent to the applications. Application Programming Interface (API) provides the abstract definition of the required reliability and its realization is transparent to the applications in the presence of the failures or attacks.

Scalability The supported mechanism to provide computational resiliency should be scalable. Use of replication mechanisms and local area network as an interconnection network can prevent the system from scalability. Overhead associated with replication and network communication should be reduced to make the system scalable, which can be achieved by load balancing.

Portability The distributed computing environments consist of wide range of computers ranging from shared memory multiprocessors, distributed memory multicomputers, to a cluster of workstations. The developed software library should be portable to these various computing systems efficiently recognizing the underlying hardware capability for optimized implementation.

1.3 Contribution

The contributions of this research are:

1. A novel approach to provide fault tolerance and automatic recovery from attacks and failures.
2. A flexible software architecture that is application and platform independent.
3. Heterogeneous load balancing of replicated structures for performance improvement.
4. Demonstration of technologies using typical real-world applications in various fields.
5. An associated analytical model expressed in terms of application-dependent parameters and resiliency requirements.
6. Experimental studies to reveal the associated overhead for computational resiliency.

1.4 Metrics of Success

The following matrices are used in assessing the quality of the suggested approach in this thesis:

1. *Overhead of Resiliency*: Investigation of the overhead of replication and how to reduce it by means of load balancing.
2. *Overhead of Recovery*: Investigation of how fast the system can recover from failure and attacks and how to reduce the overhead of recovery process.
3. *Accuracy of predictive models*: Investigation of how accurately the analytical model can perform when the number of processors, application dependent factors, reliability factors, etc. are varied.

1.5 Overview

This thesis consists of seven chapters.

Chapter 1 provides an introduction to the thesis.

Chapter 2 provides background material and related research in the fields of fault tolerance, parallel and distributed computing, and performance modeling.

Chapter 3 describes the prototype implementation methods of computational resiliency.

Chapter 4 presents the application of computational resiliency to a prototypical application, concurrent sonar processing over a homogeneous testbed, and its analytical model. Experimental study and performance prediction are also described.

Chapter 5 presents another application, remote sensing. Same experimentation and the analytical model are described.

Chapter 6 extends the applications presented in Chapter 4 and 5 to a heterogeneous system. Various load balancing methods and corresponding experimental results are described.

Chapter 7 describes the directions of future research and contains concluding remarks.

Chapter 2 Related Research

This chapter presents background study and related research in the fields of fault tolerance, heterogeneous resource management, message passing model, and performance modeling. The first section discusses the taxonomy of the fault tolerance techniques, related issues, and survey of the existing systems. The second section examines the various approaches to balance the utilization of the processors and load balancing techniques that are aware of reliability of the distributed systems. The third section describes the message passing programming model and three frequently used message passing tools. The fourth section presents the various approaches to performance modeling.

2.1 Fault Tolerance

Generally, fault tolerance means the system's ability to tolerate the failures of the system in order to complete the mission assigned transparently. Without the fault tolerance, even a single processor failure can cause the entire application running on the parallel and distributed computing environment to stop and restart from the beginning. In most of the distributed applications, fault tolerance is highly desirable for commercial applications, i.e., distributed banking systems and E-commerce servers, for mission-critical applications, i.e., nuclear power plant control and military command and control, and scientific applications, i.e., long running weather simulation applications. Fault tolerance

and recovery techniques can be implemented in hardware, software, or a combination of both. Here we are concerned primarily with software based techniques that can be applied to distributed real-time applications. Fault tolerance researches have focused on different aspects of the distributed systems, thus used different acronyms to emphasize those aspects. In this subsection, we investigate the taxonomy of the fault tolerance research and the related issues.

Capability of the fault tolerance techniques can be classified according to the types of failures they can handle. The types of failures can be categorized as three basic models [Schneider 1984, Lamport 1982]. In fail-stop failure model, in response to a failure, the component changes to a state that permits other components to detect that a failure has occurred and then stops. In omission and timing failure model, the component fails by not responding to an input or by giving an untimely response. In Byzantine failure model, the component fails by exhibiting arbitrary and even malicious behaviors, perhaps involving collusion with other faulty components. It is even impossible to tell whether a component is faulty or not.

Any system that can tolerate Byzantine failure would be the most fault tolerant. Providing more complex failure models requires higher expenses. A system is told to be *t*-fault tolerant if it can detect and mask t failures. For fail-stop failure model, $t+1$ fold replication is required to detect and mask the failures. With t failures, the $t+1$th replica can continue the operation. But, for Byzantine failures, $2t+1$ fold replication is needed. When t processors become faulty and produce faulty outputs, another $t+1$ correct

processors are required to decide the correct outputs among them. Depending on the target environments and the application requirements, the supported failure model should be selected.

A useful taxonomy of recovery techniques for information warfare has been developed by Jajodia [Jajodia 1999, Resnick 1996]. Recovery process involves reinitializing the system and replacing the failed components. Depending on the degree of transparency of a recovery process to the applications, recovery techniques can be categorized as cold-start, warm-start, and hot-start. Cold-start recovery involves a complete restart in the event of a severe attack or failure. No previous state of the system is available and the system has to restart from the beginning. The recovery times are the slowest. Warm-start involves non-transparent but automated recovery. Some knowledge about the previous state of the system is available and the system can start from the last known state. Fault detection times takes as much as cold-start, but the recovery times are lower than cold-start because of the partial initialization and state sharing. Hot-start technique is by far the more sophisticated and provides transparent recovery. Failure is completely masked and the recovery is immediate. Hot-start strategy is desirable for real-time distributed systems, leading to the fastest recovery times to the failures.

Recovery techniques can be also categorized depending on the direction of recovery, roll-backward and roll-forward recovery techniques [Randell 1979]. In roll-backward recovery techniques, when the failure happens, the system rolls back to the previously saved state and starts from there. The last available state can be saved in stable storage

for later references periodically. In contrast to roll-backward recovery, roll-forward recovery schemes always advance the sate of their work even in the presence of the failure. Usually, roll-forward recovery techniques are desirable since the system can continue the operation transparently without interruption in the presence of the failures while roll-backward techniques cause the system to interrupt the current operation and start from the past state. However, roll-forward recovery techniques may incur more overhead to maintain the alternative processes consistent and synchronized.

Most of the fault tolerance techniques developed to date are based on notion of *process replication* to provide high levels of system availability [Guerraoui 1997]. Unfortunately, the use of replication introduces additional problems such as the need to maintain consistency between replicas, detect the failure of a compromised process, and transparently recover system function. In many client-server style applications, the techniques employed to provide recovery can be divided into two general categories based on *passive* [Budhiraja 1992] or *active* [Schneider 1990] replication.

In passive replication (primary-backup aproach) [Budhiraja 1992], there is a single primary source and all other replicas are maintained purely as backups. Only the primary source receives requests from clients and guarantees the ordering and atomicity of message delivery. Although easy to implement, this method is slow to transfer control to a backup in the event of failure; this can lead to significant degradation in system response.

In active replication (state machine approach) [Schneider 1990], all replicas have the same level of control. Any viable replica may receive a message from a client and collectively the replicas maintain message ordering and atomicity. This approach is attractive for real-time systems because it provides a more transparent view of the system to client processes and is relatively fast to transfer control in the event of failure [Sussman 1996].

Two most popular implementations of software-based fault tolerance techniques are checkpointing and group communication that operate through a combination of above techniques. Checkpointing techniques can be characterized by warm-start, roll-back recovery, and passive replication. Group communication is an approach based on hot-start, roll-forward recovery, and active replication.

Group communication approach is based on replication strategy. To implement replication it is useful to organize processes into *groups* and provide communication mechanisms between groups. The concept of a process group was first introduced in the V-kernel to express one-to-many communication structures [Cheriton 1985]. A group is a set of processes sharing common application semantics, as well as the same group identifier and multicast address. Each group is viewed as a single logical entity hiding its internal structure from other groups. The processes in a group cooperate to provide a single service. In order to maintain and share a consistent process state, the processes use multicast communication primitives that guarantee every process in the group receives the same messages in the same order. The group concept has been extended to many

fault-tolerant distributed systems such as Isis [Birman 1994], Horus [Renesse 1996], Transis [Amir 1992], Totem [Agarwal 1994], and Ameoba [Kaashoek 1993].

Horus system [55] provides a flexible group communication model to the application developers. It provides an architecture whereby the protocol supporting a group can be varied, at runtime, to match the specific requirements of its application and environment. Group communication support is provided by stacking protocol modules that have a regular architecture, and in which each module has a separate responsibility. Basically, Horus supports the virtually synchronous execution model introduced by Isis[51].

Transis system [56] considers the problems that arise in diverse network setting such as network partitioning. It provides a larger-scale multicast service to solve the problems. For network partitions, Transis provides tools for recovery from them and describes how different components of a partitioned network can operate autonomously and then merge operations when they become reconnected.

Totem system [57] supports a reliable, totally ordered multicasting service over local area network and exploits the hardware broadcasts to achieve high performance. Totem is intended for the application where fault tolerance and real-time performance are critical. The characteristics of Totem include high throughput and low predictable latency, rapid detection and recovery from faults, systemwide total ordering of messages despite the network partitioning, and scalability of the underlying networks.

Ameoba [59,60] is a distributed operating system based on client/server model and uses the group communication to provide the fault tolerant operating system services, such as distributed directory service, to the users transparently. In order to tolerate the arbitrary faults, group communication is used within the distributed operating system. Group communication protocol in Ameoba uses hardware multicast capability, if on exist, for the application that needs high performance.

These systems all allow members of a group to fail thereby providing graceful degradation of performance to the point of system failure. Although not used for fault-tolerance, the process group has also been used widely as a concurrent programming paradigm through libraries such as PVM [Sunderam 1990] and MPI [Gropp 1995].

On the other hand, checkpointing is usually referred to the method to save the intermediate state of the system, *checkpoint*, in the stable repository such as hard disk or a separate server periodically. When the failure happens, the system restarts from the last saved checkpoint. Checkpointing generally requires more time to recover than process group approach since it involves restoring previous state and launching a new process.

There are tow approaches to implement checkpointing, synchronous and asynchronous. In asynchronous checkpointing, checkpoinits are taken by each process independently and no synchronization of the their actions are needed [Juang 1991]. Lack of synchronization leads to less overhead but when the failure happens the system has to search for the most recent consistent checkpoint among the processes. Sometimes,

processes may have to roll back to the initial state in the worst case, which is known as domino effect [Deconinck 1993]. In synchronous checkpointing, all the processes involved in checkpointing coordinate their actions to maintain the consistency of the checkpoints in the system. Koo and Toueg [Koo 1987] proposed an algorithm that uses synchronous checkpointing and roll-back recovery. Their algorithm solved the problem of domino effect problem that may happen in asynchronous checkpoinitng. It can also tolerate the failures that occur during the checkpointing with use of a two-phase commit protocol.

Usually, asynchronous checkpointing takes less times for checkpointing actions but may lead to unpredictably long recovery times. Synchronous checkpointing incurs more overhead for checkpointing actions but less recovery times. Therefore, if failures rarely happens, synchronous checkpointing technique places additional burden on the system [Singhal 1994].

In efforts to remove the domino effect in asynchronous checkpointing and reduce the recovery time, checkpointing with message logging approach was presented [Johnson 1989]. In this approach, the messages received are logged in the stable storage as well as the normal checkpoints. When the failures happen, a failed process is restored using the previous checkpoint and the log of messages received by that process after the last checkpoint and before the failure. With use of message logging in checkpointing, each process can be checkpointed infrequently, and no global coordination is required during execution. Checkpointing mechanisms can sometimes be used transparently and a variety

of techniques have been developed to reduce the associated overheads [Plank 1995, Ramkumar 1997, Scales 1996].

Choosing the right recovery techniques depends on the requirements of the applications. Scientific parallel programs may choose cold-start and roll-back recovery techniques. However, life-threatening applications like command control applications should choose a hot-start and active replication based strategy. This may increase the requirements for the systems for replicated servers, however, it guarantees the highest level of reliability and the fastest recovery time.

2.2 Heterogeneous Resource Management

The use of networks of personal computers, workstations, and symmetric multiprocessors as a computing platform requires load balancing techniques. Computers in a typical network often differ in processor performance, memory characteristics, and operating system. Basic concept of load balancing is to transfer load from heavily loaded processors to idle or lightly loaded processors. Many load balancing techniques over parallel computers and distributed multi-computers have been developed [Heirich 1994, Kumar 1994a, Li 1997, Watts 1998a]. These typically assume that attacks or faults are unlikely and focus on the optimal allocation of resources.

Load balancing algorithms can be characterized as static or dynamic [Shivaratri 1992]. Static load balancing decides the allocation of the workload to the system before the execution [Barnard 1994]. The algorithm cannot cope with the changes in the system at

run time. However, dynamic load balancing is done while processes are in running state and can adapt to the changes dynamically [Cybenko 1989, Evans 1993, Watts 1998a]. Dynamic load balancing algorithms, on the other hand, causes more overhead to collect and analyze the system runtime state information continuously.

Load balancing techniques are also required for fault tolerant distributed systems, especially those based on active replication strategy. Use of active replication strategy requires the processes to be replicated on several processors. Unbalanced utilization of the processors leads to degradation of the overall performance. Literature on load balancing techniques for efficient allocation of the replicated processes can be found in [Nieuwenhuis 1990, Kim 1997, Bannister 1983, Shatz 1992].

Nieuwenhuis [Nieuwenhuis 1990] has proposed a static model to represent the reliability of a replicated processes and the transformation rules that derive an optimal allocation of the replicated processes from an allocation of nonreplicated processes.

Bannister [Bannister 1983] has presented an algorithm that balances the load of replicated processes over a homogeneous system and subsequently analyzed the performance of the algorithm. An upper bound of error is provided for their heuristic algorithm. In this model, no explicit system reliability measures were presented and they did not consider the failures of the communication links.

Schatz [Shatz 1992] proposed a model that expresses the reliability of the system in terms of the probability that the system can run an entire task successfully. This model introduces a process allocation algorithm that maximizes the reliability over heterogeneous systems. The model uses a cost function to represent the unreliability caused by execution of modules on processors of various reliability, and the unreliability caused by interprocessor communication. They converted the problem of task allocation problem into state search problem and applied A* algorithm [Nilsson 1971] to obtain the optimal value of the cost function.

Kim [Kim 1997] studied static load balancing techniques for fault tolerant multicomputer systems using passive replication model. Their model is to find a static process allocation algorithm that balances the CPU load of every processor in the fault-free situation and also balances the CPU load in the presence of the failure. To avoid expensive searching time for the optimal allocation solution, they proposed a heuristic algorithm to find a sub-optimal solution.

2.3 Message Passing Model

Message passing is one of the parallel programming paradigms used widely on certain classes of parallel machines, especially those with distributed memories depicted in Figure 2.1. Several systems have been developed to demonstrate that a message passing system can be efficiently and portably implemented [Geist 1994, Gropp 1995, Taylor 1996].

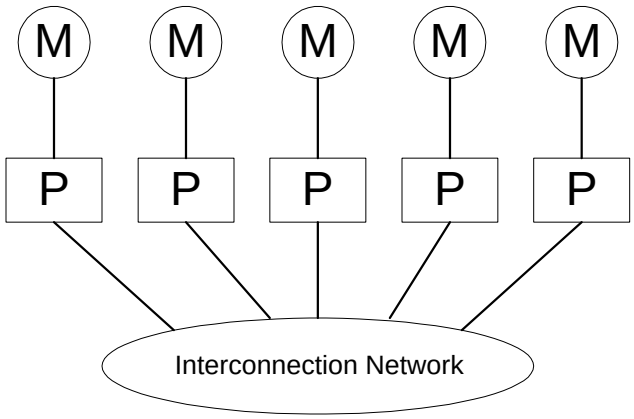


Figure 2.1: Distributed Memory Architecture

Message passing paradigm is favored for its portability. Programs developed in message passing paradigm can be ported to wide range of parallel and distributed architectures including shared memory multiprocessors, distributed memory multiprocessors, and network of workstations. It also supports various types of parallelism. It is suitable for both multiple instruction multiple data (MIMD) and single instruction multiple data (SIMD) style parallelism.

In the message passing programming model, one parallel program consists of several sequential programs that run on each processor. Each sequential program uses message passing to synchronize with and access memory contents of other processors. Each sequential program can run the different stream of instructions or the same instruction stream. The mechanism of how the messages are formatted, how they are transferred to communication devices, and how they are sent across the network should be transparent to the applications. There is a wide variety of message-passing libraries available in most of the distributed architectures. These libraries provide the applications with the

capability to run on distributed architectures initializing and managing the communication environment. Common functions provided in those libraries include synchronization, point-to-point communication, broadcast/scatter data, and gathering data from a group of processes. Three frequently used message passing libraries are described as follows.

The Parallel Virtual Machine (PVM) is a message passing tool that supports the development of parallel and distributed applications for a collection of heterogeneous computing elements [Geist 1994]. This tool was designed and developed by the Oak Ridge National Laboratory. The intention of this message-passing library is to create a single virtual machine using a group of heterogeneous computers. The library is supported by a wide variety of machines from MPPs to PCs. PVM selects a computing element to run a process on by using a process description file. Each process description file consists of a list of execution program names, locations, object file locations, and architectures. Each process is initiated in the virtual machine by the PVM daemon and spawned to the requested machine. Each task will have its own unique identifier by which it is referred to when communication requests are made. In PVM, some levels of fault tolerance are provided. The processes can ask for notification when other processes are abnormally terminated. This information can be used for the remaining processes to take action in recovery. PVM provides heterogeneous communication capability that allows data to be exchanged between different types of machines. The library emphasizes the portability issue and thus sometimes provides a lower performance compared to other packages [Koniges 2000].

The Message-Passing Interface (MPI) is a standard portable message-passing library designed through the cooperation of academia, government laboratories, and industries [Gropp 1995]. It provides an extensive collection of routines with which to create common communication schemes and is constructed from a group of communicators (a set of user-defined processing resources). Each task is ranked within each communicator, and the rank is used as a task's identifier. MPI provides many features intended to improve performance on scalable parallel computers with specialized interprocessor communication hardware. High-performance implementations of MPI that utilizes native communication services on specific machines have been provided. MPI implementation on top of standard Unix interprocessor communication protocols based on TCP/IP provides portability to heterogenous networks of workstations.

Scalable Concurrent Programming Library (SCPlib) is a designing and implementing effort of the Scalable Concurrent Programming Laboratory at Syracuse University [Taylor 1996, Watts 1998b, Watts 1998c]. This library provides a heterogeneous concurrent programming technology and has been applied to a variety of irregular, large-scale, industrial simulations such as particle simulations [Rieffel 1997, 1998], and continuum fluid flow solver [Watts 1998b, Taylor 2000]. The library is portable to a wide range of platforms, from distributed-memory multicomputers to networks of workstations, PC's and multiprocessors. It provides a *mobile thread* abstraction in which threads may move between processors to accommodate for changes in resource requirements (e.g. processor speed, memory, bandwidth). The communication structure

of an application is represented explicitly and can thus be changed transparently as a thread migrates. The library is based on a concurrent graph model, in which computational nodes are connected through arcs that correspond to communication paths. Each node consists of named states, named communicators, and execution threads (Figure 2.2).

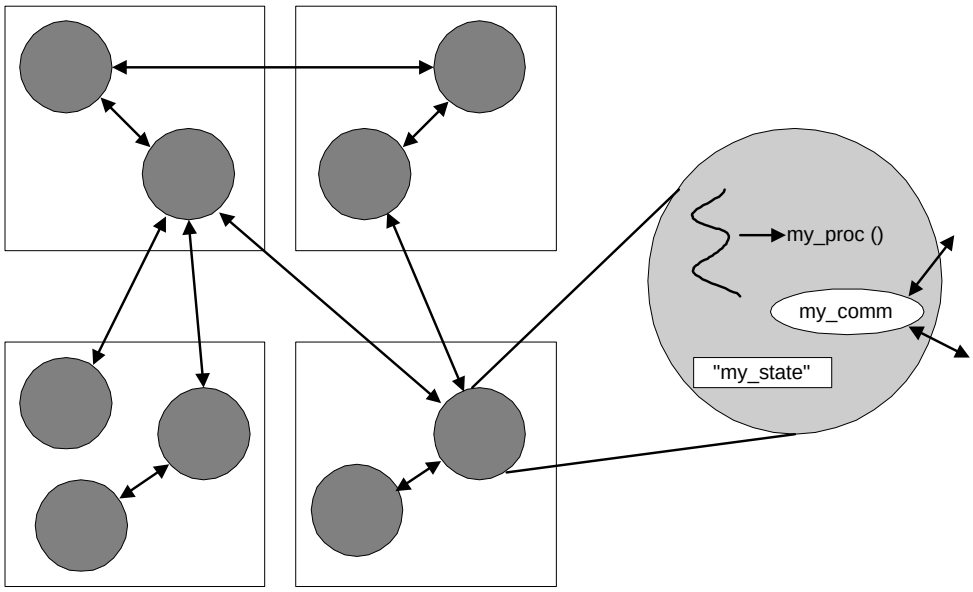


Figure 2.2: SCPlib Node Structure

The mapping of nodes to computers is transparent to the user's application, thus the library is able to move, split, and merge nodes dynamically during runtime. These capabilities enable a variety of load balancing and granularity control techniques based on thread migration. Applications based on SCPlib are MPI-standard compatible. In contrast to MPI and PVM message passing systems, SCPlib supports multithreading in its computation model. Computation model supported in SCPlib can easily fit in shared memory multiprocessors and distributed memory multiprocessors with multithreading.

2.4 Performance Modeling

Predicting the runtime of a parallel program is useful for determining the optimal values for the parameters of the implementation and the optimal mapping of data on processors[Rugina 1998]. It is also useful for analyzing the scaling behavior of parallel programs. However, deriving an explicit formula for the running time of a certain parallel program is a difficult task.

The metrics by which we measure performance can be as diverse as execution time, parallel efficiency, memory requirements, throughputs, design costs, hardware costs, portability, reliability, and so on [Foster 1994]. The relative importance of these diverse metrics will vary according to the natures of the problem at hand. For example, the design specification for reliable distributed real-time systems may specify maximum execution time and hardware costs within theses constraints.

Many techniques have been developed to predict the performance of parallel programs. They are statistical model, simulation model, analytical model [Fahringer 1996, Serrano 1994, Rugina 1998, Reiffel 1998]. All of these techniques have their own strengths and weaknesses, and their own classes of applications for which they are particularly suited. These models can be used to compare the efficiency of different algorithms, to evaluate scalability, and to identify bottlenecks and other inefficiencies, before we invest substantial effort in an implementation.

Statistical model is a performance prediction method based on statistical and probability theory. Distribution functions, random variables, probability theory, stochastic processes, Markov processes and chains, queueing networks are used to model the performance of parallel machines and programs [Fahringer 1996, Serrano 1994]. This model presents input and output data of parallel systems, analyzes the performance indices, and interprets these data by closed form formulas or distribution functions. The drawback of this model is that it is difficult to validate the statistical performance estimators against a real problem because of many simplifying assumptions.

The simulation model uses simulation techniques such as Monte Carlo, Petri net, and emulation [Baer 1990, Rugina 1998]. This model can simulate the execution of a parallel program without actually executing it on the system. This model can be used to analyze and estimate the performance of a complex system. However, it also has some problems. A detailed model requires many input parameters to describe certain aspects of both target machine and program, which may not be always available. Simulations are slow and requires a lot of computing resources[Fahringer 1996].

The analytical model is primarily concerned with predicting the performance and resource scaling characteristics on a variety of architectures. Analytical modeling techniques abstract the features of a parallel system as a set of parameters or parameterized functions in order to make the modeling task tractable [Meira 1995]. Advantages of this method include that it is usually inexpensive and provides abstract view of the hardware and software, and that it is easier to tune and validate the accuracy

of individual parameters. However, the methods are usually not accurate when compared to real executions due to simplifications in the modeling process. For example, it is unclear how to model dynamic behavior such as adaptive load balancing and the changes of network traffic loads. Examples of analytical models developed for real systems can be found in a particle simulation model [Reiffel 1998].

Each model described has its own advantages and disadvantages. Choosing among the models is highly application-dependent. In our work, we are primarily interested in predicting the maximum execution time of distributed real-time applications that can be expressed as application dependent parameters, and the required system resources for various reliability parameters on a variety of computer architectures. We thus use the parameterized algebra-based analytical model for prediction.

2.4 Summary

In this chapter, we have surveyed and investigated various approaches to build scalable and fault tolerant distributed applications. Approaches to be taken are highly dependent upon the characteristics of target applications. For those applications as distributed real-time applications, that this thesis is targeted for, following strategies are desirable for:

- **Fault Tolerance and Recovery Techniques**

  Hot-start, roll-forward, and active replication based fault tolerance and recovery techniques are desirable to provide faster response and recovery time. However, use of these techniques may make design and development more complicated and difficult.

- **Heterogeneous Resource Management**

  Dynamic load balancing techniques are desirable to cope with the dynamic changes of the distributed computing environments.

- **Concurrent Programming Paradigm**

  Our approach uses the Scalable Concurrent Programming Library (SCPlib). This library provides a basic concurrent programming technology on a wide range of platforms**.**

- **Performance Modeling**

  Analytical performance modeling methods are used to predict the runtime of the applications.

Chapter 3 Computational Resiliency

Computational resiliency provides higher level of fault tolerance by sustaining operation and dynamically restoring the level of assurance in system function in response to an attack and failure. This approach is intended to assure that full operational readiness and robustness of the system are restored within a quantifiable finite time, subject only to the constraints imposed by available resources. This chapter describes the concepts of computational resiliency and its prototype implementation. Software architecture to realize the concepts is also presented.

3.1 Introduction

To tolerate information warfare (IW) attacks, applications may choose to statically replicate mission critical threads, thereby forming *thread groups*, as shown in Figure 3.1. Each thread in a group is allocated to a different computational resource so as to sustain operation. This provides a graceful path of performance degradation to the point of failure. Unfortunately, it is not resilient in that it does not *assure* continued operation of the system when sufficient resources are available elsewhere in the network. In any realistic system, there will never be sufficient resources to replicate all threads, therefore some policy-based methods for controlling replication are required. In addition, resources may become available, or unavailable, dynamically, during the course of a conflict.

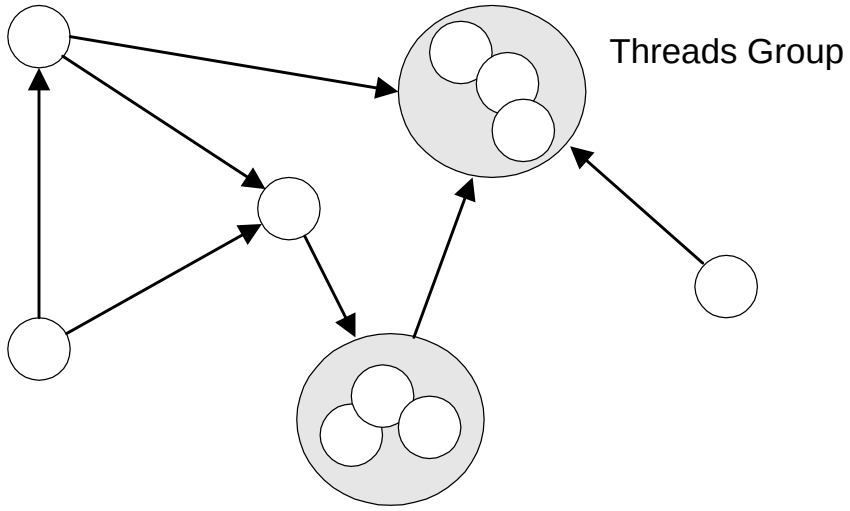


Figure 3.1: Replication of Threads

An alternative approach is to *dynamically recreate the level of thread replication* in the face of attack. This assures that operational readiness is eventually restored, subject only to the constraints imposed by the time-dependent availability of resources. Obviously to be successful, the replacement thread must be mapped to an alternative location in the network with sufficient resources. Protocols are required to dynamically reconfigure communication between residual thread groups and newly created replicas. These protocols deal with race conditions inherent in the reconfiguration process, ensure that no communication is lost, that the integrity of state is maintained, and that where possible locality of communication is preserved.

Figure 3.2 shows how resiliency is layered into a distributed application. The application programmer simply describes the required thread structure and states the level of resiliency for each crucial thread. In the diagram there are three threads, the first and second are resilient to level three, while the third is resilient to level two. Communication

between threads at the application level is replaced by group communication at the resilient level. Threads are subsequently mapped, through indexing, to appropriate processors such that replicas in a single group are placed in different processors at the architectural level.

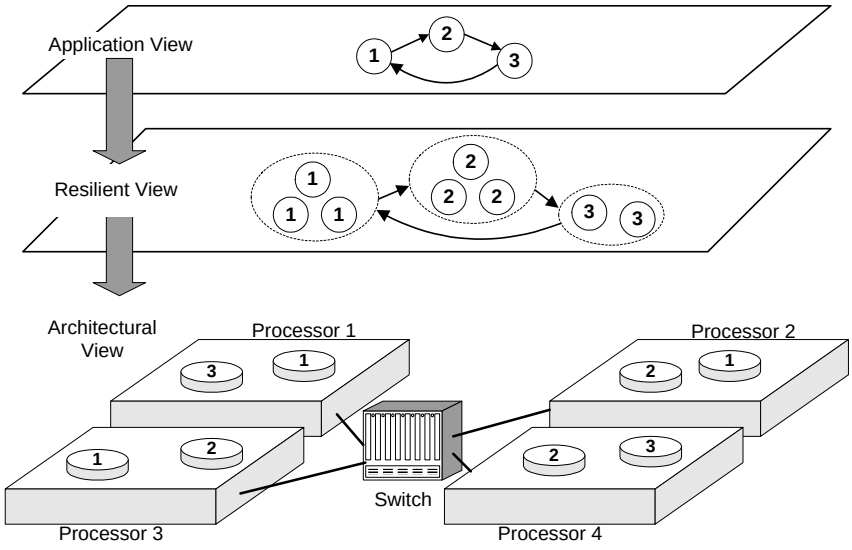


Figure 3.2: Computational Resiliency Using a Cluster of Multiprocessors

Figure 3.3 compares the fault-tolerant model of computation with computational resiliency. In a fault-tolerant implementation (dashed line), as threads are compromised, graceful degradation occurs and eventually, when no replicas are available, the application is unable to proceed.

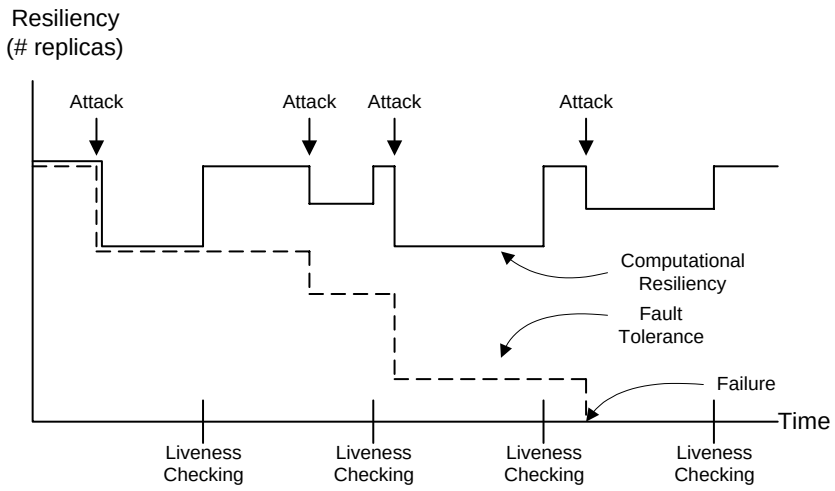


Figure 3.3: Fault-Tolerance vs. Computational Resiliency

Using resiliency, periodic liveness checks are performed. These checks are *not* designed to detect an IW attack, but rather, they seek to determine if an application is not performing as expected. If an application thread is detected as compromised during a liveness check, it will be destroyed and replaced using the uncompromised residual members of the group. This hot-start recovery mechanism [Jajodia 1999] ensures that the newly recreated thread begins execution from the most recent state rather than a state where the compromise occurred. No message logging or intermediate state is saved either in stable storage such as a hard disk, or at a remote server. Therefore, network file system failure does not affect robustness.

3.2 Prototype Implementation

To provide highly mobile threads with the ability to reconfigure and replicate in a heterogeneous computing environment, it is necessary to have an explicit representation

of the *communication structure* used by the application. We have developed a concurrent programming library that provides this basic functionality [Taylor 1996, Watts 1998c, Watts 1998b]. Distributed applications are composed of a collection of threads that communicate and synchronize either through shared memory or by sending messages. Each thread has an associated *state*, which is operated on by application specific routines e.g. in a remote sensing application this may involve matrix algebra for image manipulation. A thread also has a machine independent description of its communication structure. In general these systems are *reactive* [Seitz 1985] in that the important transitions between data states occur at the receipt of messages. This provides a natural mechanism to synchronize each thread, detect an information warfare attack, and initiate appropriate recovery.

To dynamically recover replication, a mechanism is required to recreate a compromised thread with the appropriate communication structure at a new location in the network. This mechanism is implemented by replicating a residual thread from the compromised group and subsequently moving the new thread to its desired location. Unfortunately, it may not be efficient, either because of memory disparities or thread *granularity* (i.e. ratio of computation to communication) to move the new thread directly. Thus additional mechanisms are needed to allow thread granularity to be increased, by merging, or decreased, by splitting, the associated computation. Armed with these basic techniques: thread *move*, *merge* and *split*, it is possible to build concepts for resource management [Watts 1998b]. These basic operations are outlined in Figure 3.4. There exists no general solution to the resource management problem, thus each application must employ an

appropriate technique [Bokhari 1981]. For simplicity, in this thesis a Manager-Worker approach is used to demonstrate the ideas [Chandy 1992].

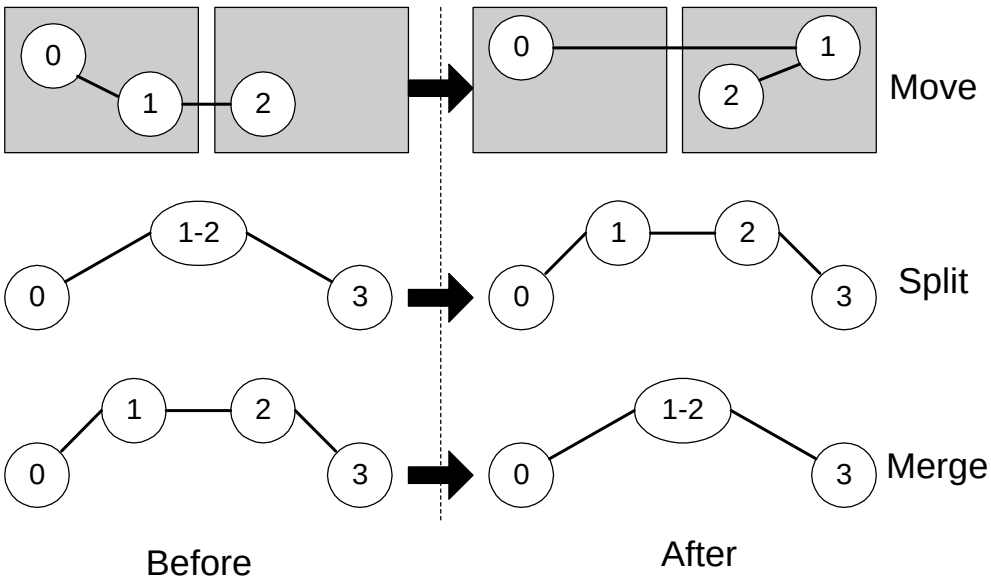


Figure 3.4: Resource Allocation and Mapping

The crucial issues associated with use of a dynamically reconfigurable group of replicated threads include describing and managing the group, detecting a compromise, and ensuring valid program state and communication structure. From a programming perspective we seek to hide the implementation details associated with these issues in a programming library and so relieve the application programmer from the complexities associated with resiliency. A programmer may simply specify the level of resiliency (i.e. number of thread replicas required) when initially spawning a thread. This level of resiliency will then be maintained automatically throughout the runtime of the computation provided that there are sufficient resources. Resiliency will gracefully degrade when resources are stretched beyond their capacity. The programming library

implements three protocols that address these crucial issues by providing group membership, liveness checking and recovery, and flow control. We use two prototypical applications to demonstrate how those protocols are used and quantify the associated performance in the subsequent chapters. In the following sub-sections, we describe three protocols in detail.

3.2.1 Membership Protocol

The membership protocol provides mechanisms to *create* threads and cause them to *join* or *leave* a group. At the beginning of program execution, groups are constructed by creating replicas and causing each to join the group. During failure and recovery, when a thread is compromised, it is forced to leave its group; a new replica is then created that joins the group to take its place. Groups are numbered for addressing purposes and the organization of communication is keyed to this numbering. Programmers follow the standard Application Programming Interfaces (APIs) that allow them to specify the required resiliency for a thread and create communication channels between groups. At the user level, following APIs are used to specify their computation and communication structures. Here we use simplified abstract expressions for those APIs and the detailed description of each API will be presented in Appendix A.

- thread_create (*groupid*, *thread_fn*, *resiliency*)

  Creates a thread that will belong to a group with *groupid* and has a replication degree of *resiliency*.
- channel_create(*group1*, *group2*)

Creates a communication channel between *group1* and *group2*.

Program 3.1 shows abstractly how these functions are used to implement the process structure shown in Figure 3.2. Three groups are created (1,2,3), each designated by an appropriate unique identifier (g1,g2,g3). When the first group is created the programmer specifies resiliency of three, meaning three replicated threads in the group g1 (1). Similarly, the second thread has three replicas and the third has two. The mapping of threads in this example occurs abstractly through the @ notation. After groups are created, architecturally independent communication channels are created between groups (4,5,6). Communication between threads that are not replicated, i.e. resiliency 1, involves no overhead associated with group representation.

```
main () {
  g1 = thread_create (1, thread1_fn, 3) @ C1, C2, C3          // 1
  g2 = thread_create (2, thread2_fn, 3) @ C2, C3, C4          // 2
  g3 = thread_create (3, thread3_fn, 2) @ C1, C4              // 3
  channel_create (g1, g2)                                     // 4
       channel_create (g2, g3)                                // 5

  channel_create (g3, g1)                                     // 6
}
```

Program 3.1: Abstract Code for Figure 3.2

Application program doesn't start until the initial resiliency requirements specified by the user are met. Once resiliency requirements are met and the application program starts running, the computational resiliency protocols takes care of keeping those resiliency requirements until the end of the program.

Our implementation of computational resiliency depends on the notion of group [Cheriton 1985]. It is important to maintain a well-defined service semantics. Application developer can rely on the semantics when designing correct applications using this group communication service [Amir 1995]. The semantics specify both the assumptions taken and the guarantees provided. Groups supported in our model have the following semantics and properties.

Group addressing Group addressing refers to how to identify a group and how to address a group. Each group is identified with its *groupid* that is an unique non-negative integer. Users specify the *groupid* when they create a thread whenever needed.

Group locality Since we replicate threads for better fault tolerance and survivability, the replicated nodes cannot be created in the same computer. Each replicated thread is scattered around the system.

Group membership Each thread can only belong to a single group, i.e. overlapping groups are not allowed. Each member of a group is identical and serves as a replicated thread. Therefore, there is no reason to let a node belong to more than one group.

Delivery semantics Our model provides the ordered delivery of the message on per-channel basis. A thread has a named communicator that consists of ports of various types and a communication channel is established between two ports in each group. Reliable

ordered delivery of the messages between two groups is guaranteed in the presence of failures.

Open group A member of a group can send a message to other group.

Failure model The failure models supported are fail-stop, and omission and timing failure models. In these types of failure, the crashed threads just stop after crash. Crashed threads neither perform any unpredicted activities nor generate malicious messages to the other members. A group of n members is $n-1$ fault tolerant since it can tolerates $n-1$ member crashes until they are recreated.

3.2.2 Liveness Checking Protocol

The liveness checking and recovery protocol provides an interface to application specific routines for detecting a compromise and implements the recovery mechanism. The frequency of liveness checking is determined by the programmer and is application dependent. Liveness checking is performed globally across the application and the protocol uses the underlying mechanisms for thread movement to recreate compromised threads and reconfigure group communication. Since this involves both reconfiguring a group and its inter-group communication, all threads must be involved. When a liveness check occurs, the members of each group communicate, and one of the uncompromised threads is selected as the group leader. The leader then coordinates dynamic replication and changes to the group communication structure to both exclude compromised threads and include the new replicas. Bounds on latency and timeouts are used to circumvent

compromised threads that simply fail to respond during the protocol. The application programmer is simply required to determine at what point a liveness check is to be performed.

Liveness checking protocol includes four steps;

1) Detect failures

Detection of the crashed members in each group is based on timeout. Each member of the group broadcasts a *liveness checking* message to every member in the group to indicate that it is alive. Everyone listens to the broadcasted *livenss checking* messages from other members until the timeout period. After the timeout, the members that failed to notify the others of its *livenss* are regarded as crashed. After detection of the crashed members, a group leader is elected to represent the group to and represents its group during the liveness checking, which is called a Local Group Leader (LGL). Global Group (GG) is a group that has the lowest *groupid* and Global Group Leader (GGL) is the group leader of GG. In order to synchronize and coordinate the actions of each group and dissemination of crash and reconfiguration information, a hierarchy of groups is constructed. Figure 3.5 shows the hierarchical structure of the groups and members. In computational resiliency, each thread is identified with a pair, (*compid, nodeid),* that is unique throughout the entire system. A member with lowest (*compid, nodeid*) among the surviving members is selected as a group leader.

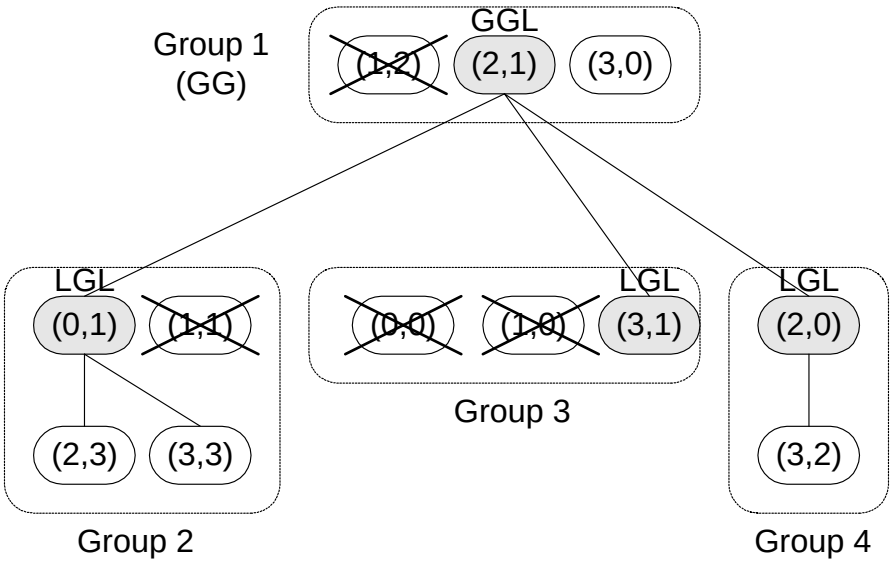


Figure 3.5: Hierarchy of Groups during Liveness Checking

Failure detection based on timeout mechanism cannot distinguish slow threads from crashed threads. Threads that run on slow processors may fail to broadcast its *liveness message* in time, which yields to omission and timing failures. In order to overcome this impossibility to distinguish slow threads from crashed threads, slow threads that caused omission and timing failures are intentionally kill by LGL. This aggressive approach may waste the computing resources but allow the system tolerate omission and timing failures. Crashed members are forced to leave the group.

2) Reconfiguration and Recreation

Each group leader reports its member status to GGL to build a global snapshot of the system. GGL collects member information of each group and broadcasts this information to every group leader and down to each member of the group. With this information, each thread removes the communication channels to the crashed threads and membership

information. Each group leader recreates crashed members at another location. Current computation states are copied to created threads from the group leader. Communication states are also copied and the actual communication links are established accordingly. In this recreation and reconfiguration process, the newly recreated threads will have the most recent computation and communication states, which prevents the system from rolling back to the previous states. Another important issue is to preserve locality of the communication. When a thread is recreated at another location, its communication states should be maintained consistently. Figure 3.6 shows that when thread 5 crashes on computer 1, it is recreated at computer 2 while preserving the same communication structures.

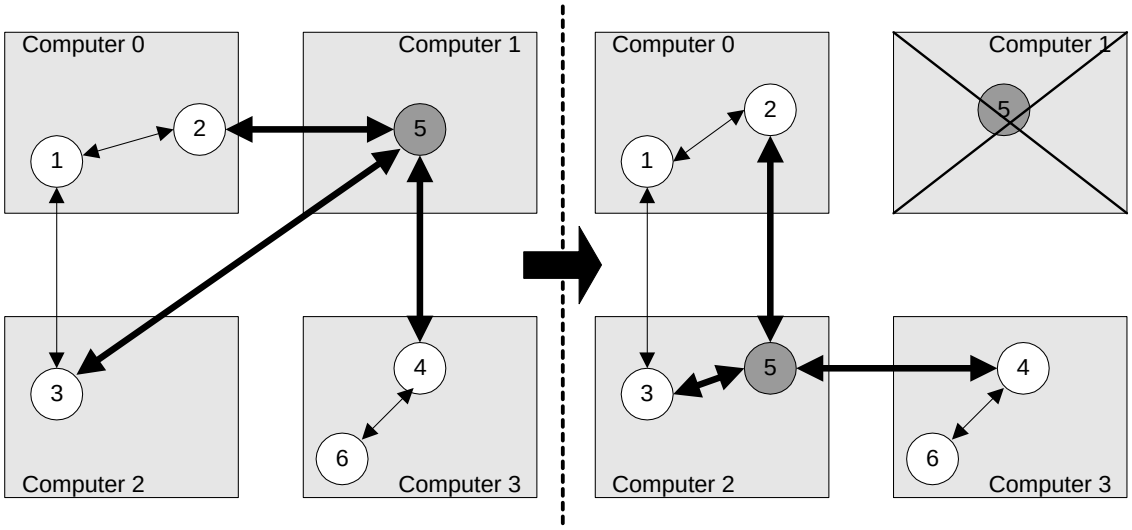


Figure 3.6: Recreation of the Crashed Thread

3) Synchronize the membership information

Exchange the information about the newly created threads in each group. After this operation every member in every group has the same view about the entire system. The

same group hierarchy is used for collecting and broadcasting the membership information.

4) Resume operation

Every thread in the system is notified that the reconfiguration is ready and resumes its operation.

For liveness checking, following abstract API is provided to the application programmers. Once again we used simplified version of APIs to represent the abstract codes.

- liveness_check (*user_states*)

  Performs livenss checking for a group

Program 3.2 illustrates how liveness checking is used. Assume that the process structure in Figure 3.2 is used to simply circulate a token among the threads. The first thread is responsible for injecting the token (1,2). The basic action of each thread is to repeatedly receive a token from the left (3), and send it to the right (4). The programmer organizes the application such that periodically liveness checking is performed (5). At that point, compromised threads are detected and recreated as long as there are available resources.

```
thread_fn (left, right) {
  if(my group_id ==1)                    // 1
     group_send(right, token)            // 2
  while(true) {
     token = group_recv (left)           // 3
     group_send (right, token)           // 4
     if (time_expired)
        liveness_check (states)          // 5
  }
}
```

Program 3.2: Abstract Code for Threads in Figure 3.2

Figure 3.7 depicts the state of the example application when either processor 3 or its network connection is compromised. As a result of this compromise, two threads, one from group 1 and one from group 2 are compromised.

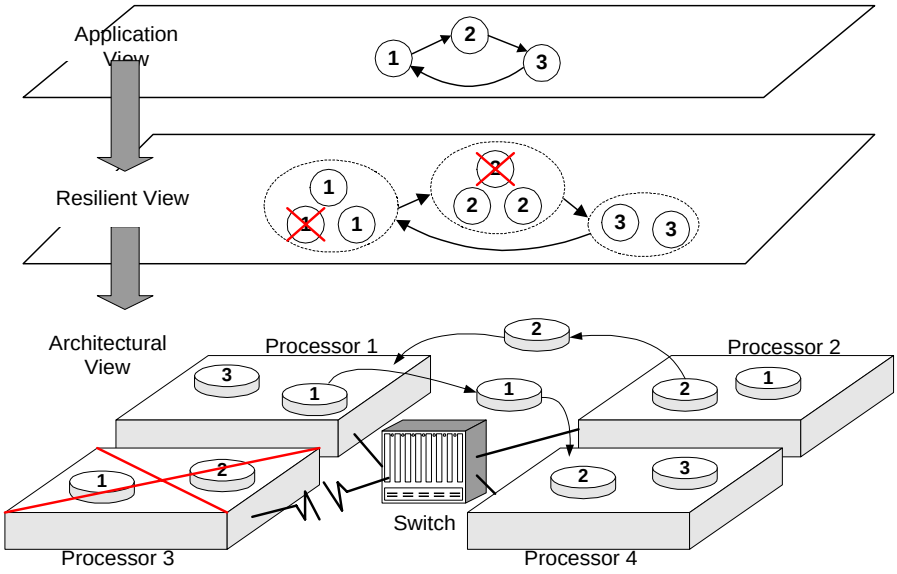


Figure 3.7: Failure and Reconfiguration

At the next liveness check, these threads are recreated at processor 1 and 4, respectively as shown in Figure 3.8. The new threads have the same computation state and communication structure as the residual, uncompromised, threads in their groups. As a result, the required level of resiliency is re-established and the application can tolerate further attacks in future. Notice that the reconfiguration of the compromised threads is transparent, as there are no changes at the application layer.

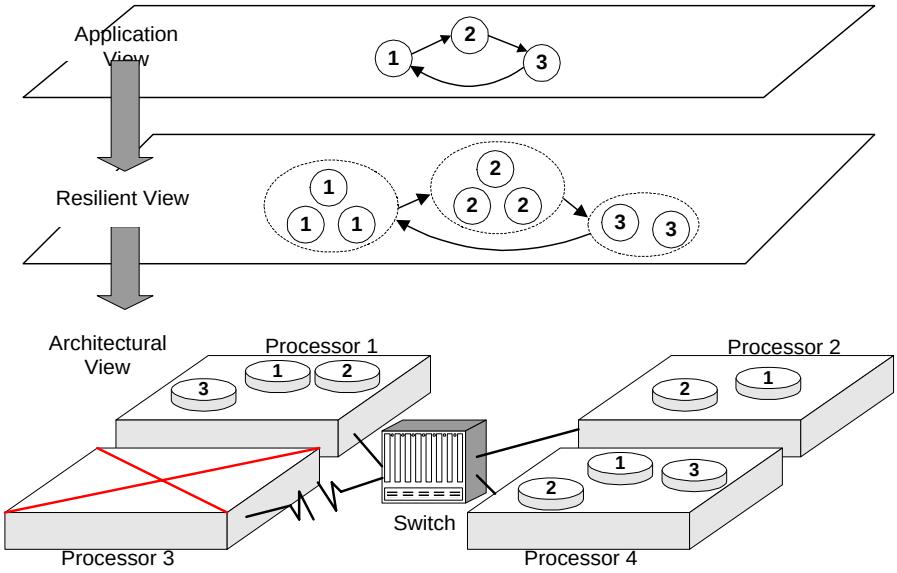


Figure 3.8: After Liveness Checking

In our approach, coordinated liveness checking method was used. Coordinated checkpointing methods prevent the entire program from rolling back to previous states and avoid domino effect [Toueg 1987]. This approach also enables roll-forward and hot-start recovery scheme, which leads to faster response to the failures. In contrast to most checkpointing approaches, stable repository or separate server is not used to retrieve the more recent state. Information warfare attacks or failures may include Network File

System (NFS) crash. Saving and retrieving intermediate states in NFS may prevent the system from retrieving the saved states on NFS failure. Another reason to avoid saving intermediate states in hard disk is that saving intermediate states or messages may take much longer time for certain applications. For example, most of client-server applications use small message size. However, remote sensing applications to be introduced in Chapter 5 may use more than hundred MBytes messages, which causes tremendous overhead for I/O operations if message logging or checkpointing is used.

3.2.3 Flow Control Protocol

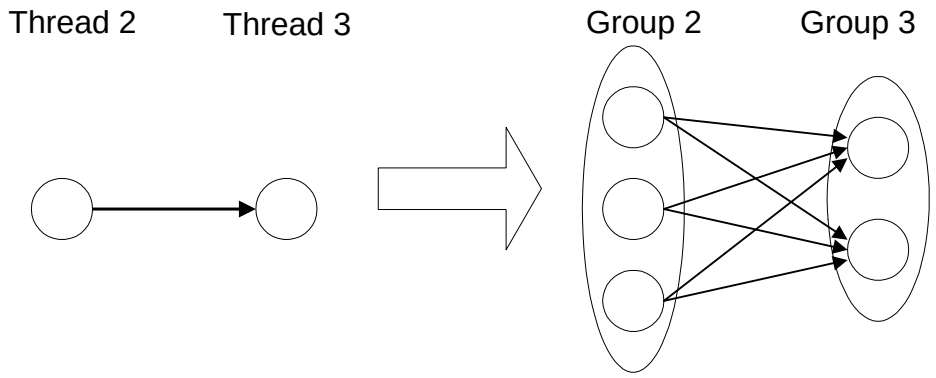
The flow control protocol ensures reliable, ordered delivery of messages between the groups in the presence of a compromise. Figure 3.9 shows the impact of replication on the communication structure in Figure 3.2. At the application layer, threads 2 and 3 communicate directly through a single point-to-point channel. At the resilient layer, the sender has replication level 3 and is represented by group 2, while the receiver has replication level 2 and is represented by group 3. The actual communication structure implemented to achieve this group communication is shown on the right. Each thread in group 2 replicates its communication to group 3. This communication is hidden from the programmer in that it is provided by virtue of the implementation of group communication. For sending and receiving messages through the communication channel, following abstract APIs are provided.

- group_send (*groupid*, *message*)

  Sends a message to *groupid*

- group_recv (*groupid*, *message*)

  Receives a message from *groupid*



(a) Application View (b) Resilient View

Figure 3.9: Group Channel Implementation

Figure 3.10 shows how the flow control protocol effects message transport. Notice that the sending group on the left has three members (resiliency of level 3) and the right hand side shows one of the receiving threads. The receiving thread may receive the same message three times without error or less than three due to compromises. In this picture, the third channel has failed and no more messages are transmitted after the fourth message over this failed channel. The receiver discards the duplicated messages, reorders the incoming messages, and delivers them to the application level thread. The shaded messages in the picture are duplicates that are received but discarded.

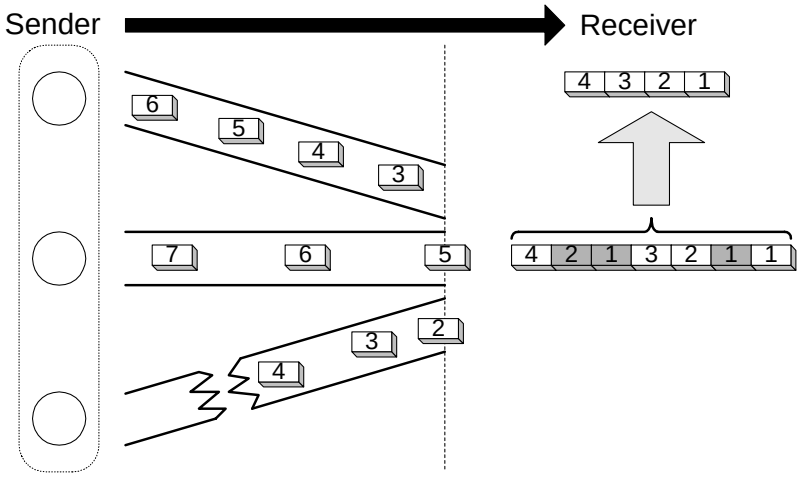


Figure 3.10: Flow Control

3.4 Software Architecture

In order to support the above functionality comprising heterogeneous distributed multiprocessing, fault-tolerance and resiliency, and resource management, a general software architecture is needed that can integrate these concepts and concerns. Figure 3.11 shows what this software architecture involve. At the bottom there is a hardware dependent layer that consists of network layer, thread layer, and device layer. Underlying networking technology may vary like Gigabit Ethernet, switched Fast Ethernet, or ATM network. Thread library layer provides various thread packages such as IRIX threads, Solaris threads, Quick, Pthreads, and Windows threads depending on the computing platforms. Device layer provides a specific device drivers for the sensors in real-time distributed applications. For example, in real-time multi-spectral image processing application, the multi-spectral image sensors are used in the system. Heterogeneous computing layer provides services to transforming the data representation that may vary on heterogeneous computing environment. Reconfigurable thread layer provides basic

primitive operations relating to computation threads and communication channels. Above that, three higher level services are provided: a secure heterogeneous computing layer, resource management layer, and resilient computing layer. Each of these layers deals with specific application concerns. Secure heterogeneous computing layer provides the necessary security services to applications. Resource management layer provides functionality to manage competing resources efficiently. Resilient computing layer provides the transparent failure masking and recovery schemes.

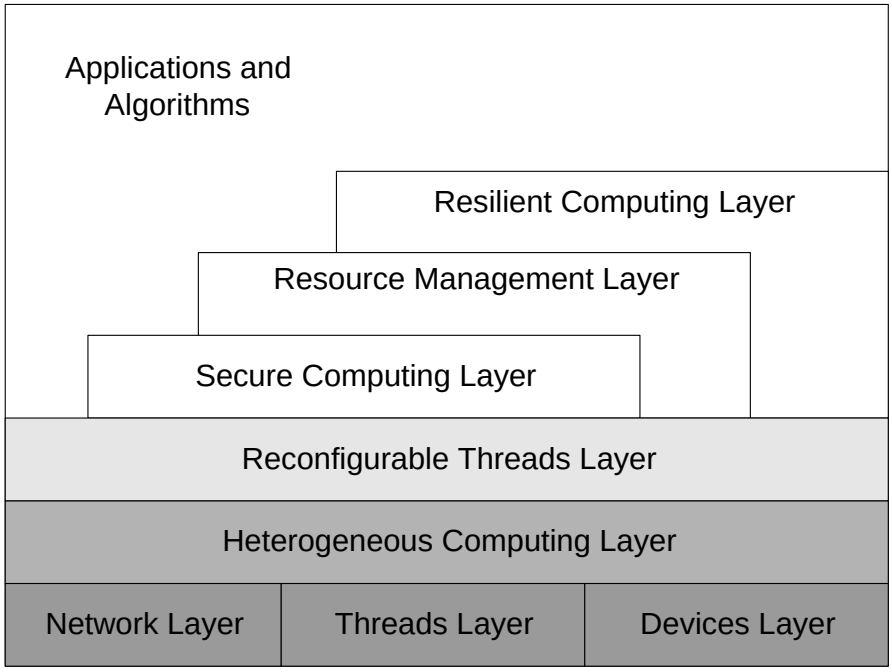


Figure 3.11: Software Architecture for Computational Resiliency

This architecture is highly reconfigurable and programmable in that different service layers can be structured dynamically to provide necessary services depending on the application's requirements. Applications can simply use reconfigurable threads layer directly for better performance in fault-free environment. For an application that runs in

hostile environment and needs fault tolerance and security, fault tolerance and secure computing layers can be used collectively to ensure desired level of security and fault tolerance. One of the topics in this thesis is how the constraints imposed by differing services affects application performance. This thesis examines the tradeoffs involved to determine analytic models that give insight into the general principles behind such architectures.

3.3 Summary

This chapter has presented the concept of computational resiliency and the prototype implementation. Computational resiliency is distinguished from the fault tolerance in that it restores the degree of replication back to the required level in the presence of failures and attacks. Computational resiliency addresses three major issues: clustered multiprocessing, resource management, and fault tolerance. Prototype implementation includes message passing for concurrent processing and three basic protocols for fault tolerance, membership, liveness checking, and flow control protocols.

Characteristics of developed prototype implementation include:

- Novel software architecture for flexible and programmable services.
- Message passing model for concurrent programming.
- Roll-forward and hot-start recovery scheme.
- Active replication to reduce the response time to the failures and attacks.
- Independent of Network File System (NFS) for recovery.
- Does not save the intermediate computation.

- No message logging to save the communication states.

Our approach may require more computing and communication resources than non-replication based approaches. However, providing faster response time and increased reliability justifies use of more resources for certain class of applications, for example, real-time distributed applications.

Chapter 4 Concurrent Sonar Processing

This chapter presents how computational resiliency can be applied to one of the prototypical applications, concurrent sonar processing. Experimental studies and an analytical model for the application are presented. These studies utilize the model to predict the scalability of the algorithm and a variety of other useful characteristics. Performance issues associated with resiliency are also investigated.

4.1 Introduction

Sonar systems detect, locate, and classify underwater targets by acoustic means [Nielsen 1991, Curtis 1980]. One of the most important processes in sonar operations is beamforming. This process combines the outputs from a number of omni-directional transducer elements, arranged in an array of arbitrary geometry, so as to enhance signals from some defined spatial locations. It also suppresses signals from other non-target obstacles. Beamformers must be capable of forming and processing large numbers of narrow beams simultaneously to give reasonable angular cover, as well as good angular resolution. In addition, beams must be independently steered and stabilized to compensate for the effect of a ship's motion.

In collaboration with the Ocean, Radar, and Sensor Systems Division at Lockheed Martin, we have developed a concurrent towed array sonar application based on conventional beamforming techniques [Lee 2001, Barnard 1998]. In this section, we

describe how to implement concurrent sonar processing application using computational resiliency. Experimental results on homogeneous systems will be discussed and the analytical model is presented.

4.2 Computational Resiliency

The general concurrent structure of this application is shown in Figure 4.1. A sensor thread is constructed to simulate the signals emanating from a towed array sonar, containing N_E sensor elements. This simulation creates the sonar returns that would emanate from a generic submarine. The 360 degrees of sonar resolution is partitioned among *M beamformer* threads. Each thread fifo-buffers N_S partial returns and repeatedly computes a covariance matrix and a partial beamforming result for the set of angles in the partition. The partial results are combined at a separate thread that performs analysis based on triangulation to determine the track and speed of the target. This thread also presents a waterfall display of the result.

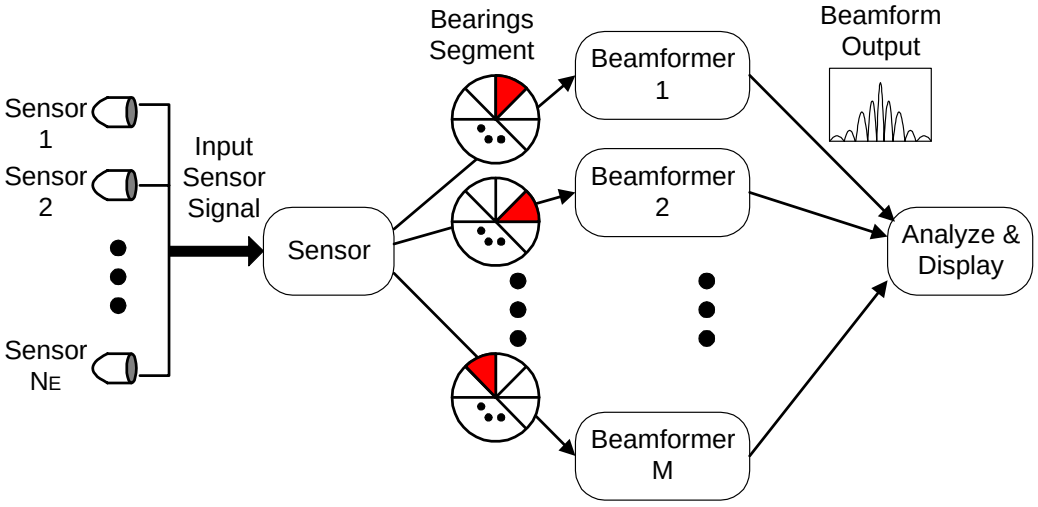


Figure 4.1: Communication Model for Sonar Processing

Figure 4.2 shows how this application is implemented with computational resiliency and shows the application layer. In this communication structure, five threads corresponding to the sensor, analysis, and beamformers, are connected through communication channels.

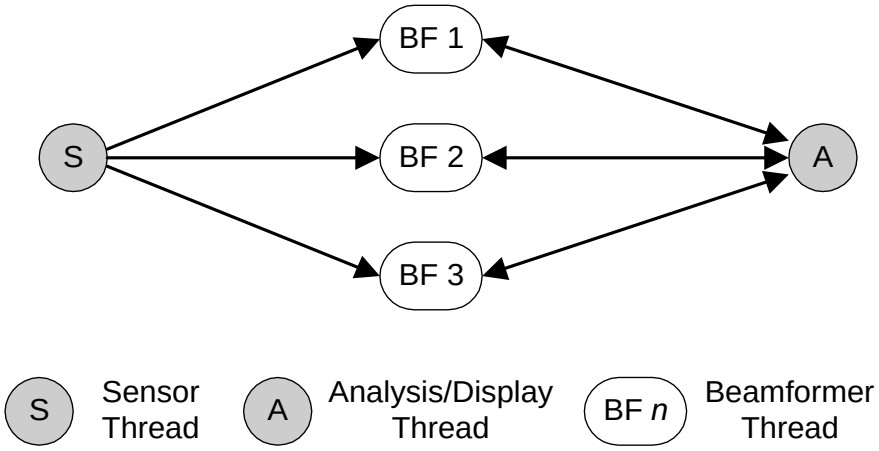


Figure 4.2: Application View

Figure 4.3 shows the resilient view of the same application in computational resiliency. In this picture, the beamformer threads are replicated with degree two.

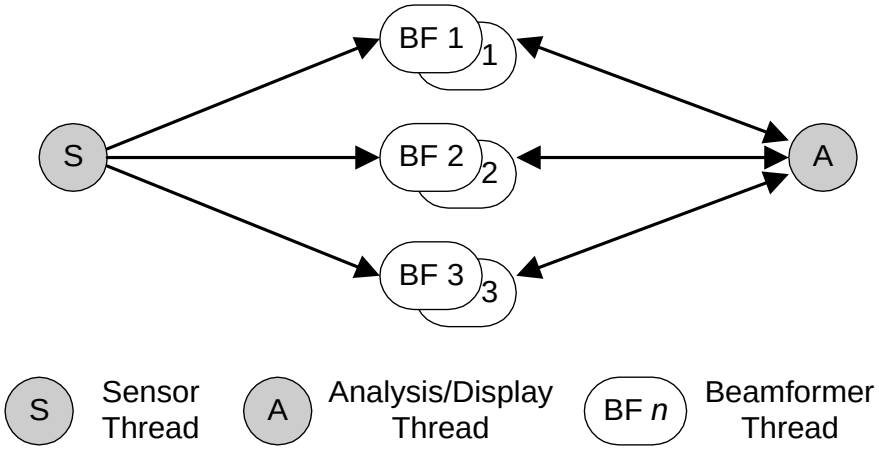


Figure 4.3: Resilient View

Program 4.1 shows the abstract code for initializing the sonar application. The user constructs the thread structure by specifying the required resiliency, one for both sensor (1) and analysis (2) threads and two for each beamformer thread (3). Each beamformer is associated with a communication channel to the sensor (4) and the analysis thread (5).

```
sonar () {
  gs = group_create (sensor_fn, 1) @ C0                    // 1
  ga = group_create (analysis_fn, 1) @ C0                  // 2
  for (i=1; i<=3; i++) {
    gbf[i] = group_create (beamform_fn, 2) @ Ci, Ci+1       // 3
    channel_create (gs, gbf[i])                            // 4
    channel_create (gbf[i], ga)                            // 5
  }
}
```

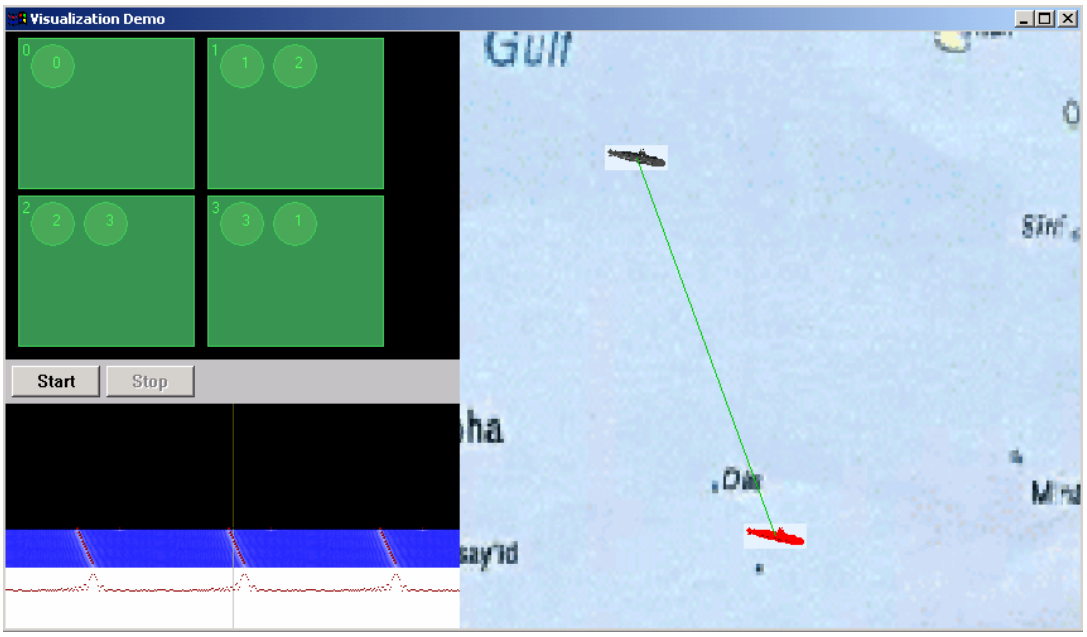
Program 4.1: Abstract Code for Sonar Initialization

Program 4.2 shows the abstract code for the beamformer threads. Sonar returns are received from the sensor (1) and stored in a fifo buffer (2). A partial covarience matrix *pcm* is then formed by each beamformer (3) and sent to the analysis thread (4). Eventually, the analysis thread responds with a complete covarience matrix *cm* (5). The beamforming calculation can then be performed to build a partial beamformed result *pbf* that corresponds to the returns processed by the beamformer (6). This partial result is then sent to the analysis module for waterfall display (7). Prior to the processing of the next set of returns, if it is the appropriate time, a liveness check is performed (8) to provide survivable operation.

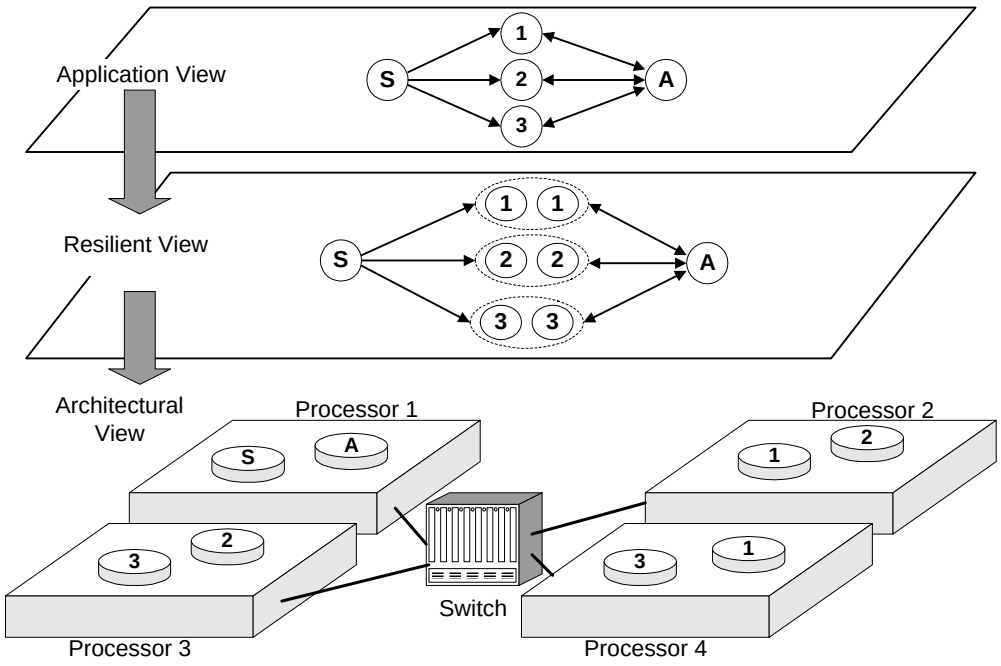
```
beamform_fn (sensor, analysis) {
  fifo = create_fifo()
  while (tracking) {
    bearing_segment = group_recv (sensor)        // 1
    add (bearing_segment, fifo)                  // 2
    pcm = covariance_matrix ()                   // 3
    group_send (analysis, pcm)                   // 4
    cm = recv (analysis)                         // 5
    pbf = beamform (fifo, cm)                    // 6
    group_send (analysis, pbf)                   // 7
    if (time expires)
        liveness_check ()                        // 8
  }
}
```

Program 4.2: Abstract Code for Beamformer Threads

Next three figures, Figure 4.4 to 4.6 show the sequence of captured screenshots of the demonstration program for the example program in Figure 4.2 and 4.3. It was developed on Windows NT/2000 platforms for simple technology demonstration. This demonstration program shows the current mappings of the threads to the computers, accumulated sonar processing results in the waterfall display, current sonar processing result for 360 degrees, source and target submarines cruising a random path in the Persian Gulf. Figures also show the corresponding three layers of views. Along the sequence of failures, application view doesn't change at all providing transparent fault tolerance and recovery to the users. Resilient view changes when a failure occurs but restores the original shape after recovery. Architectural view changes permanently as the failure happens and the recreated threads are mapped to new locations.

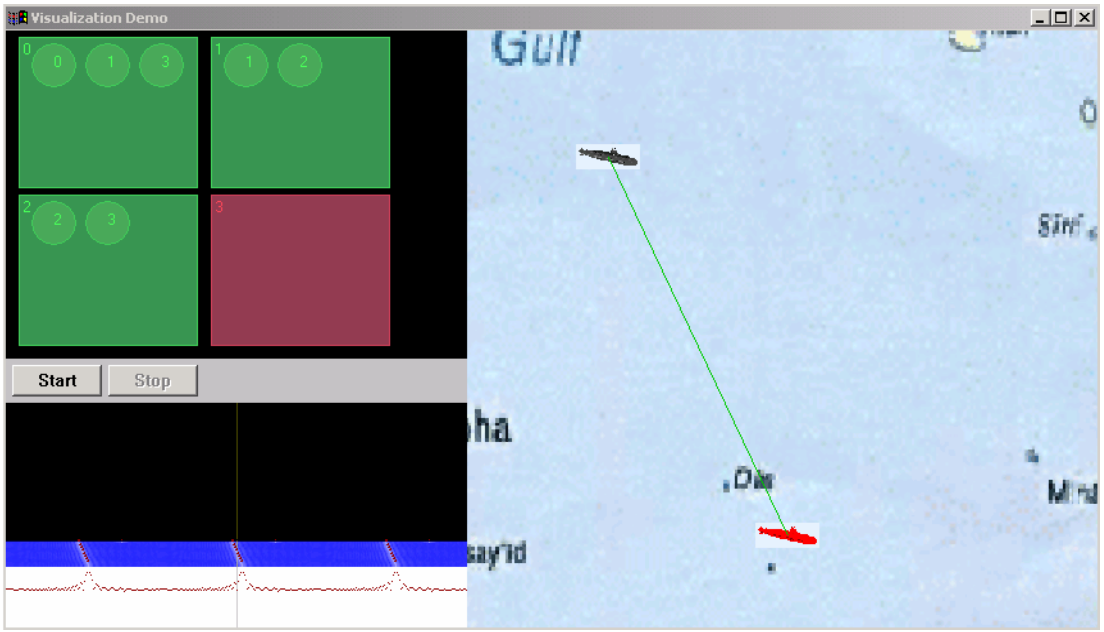


(a)

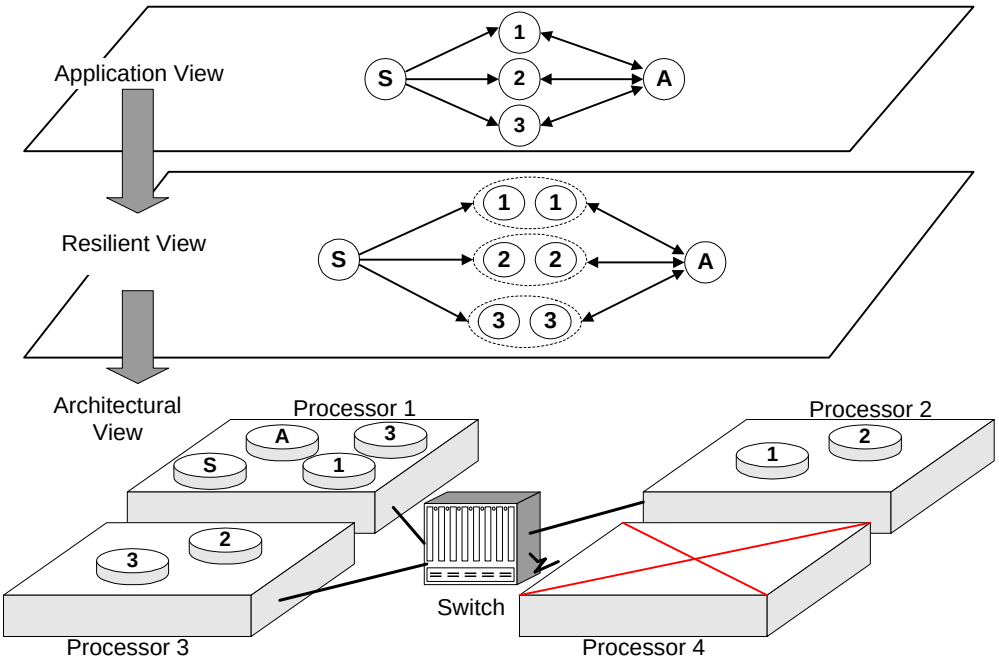


(b)

Figure 4.4: Before Failure

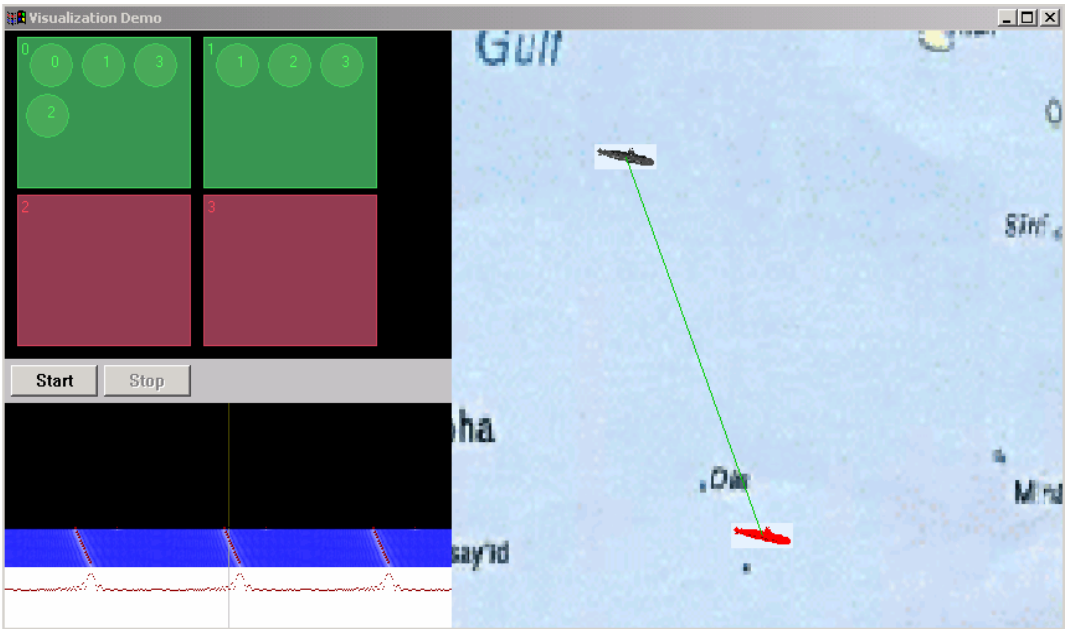


(a)

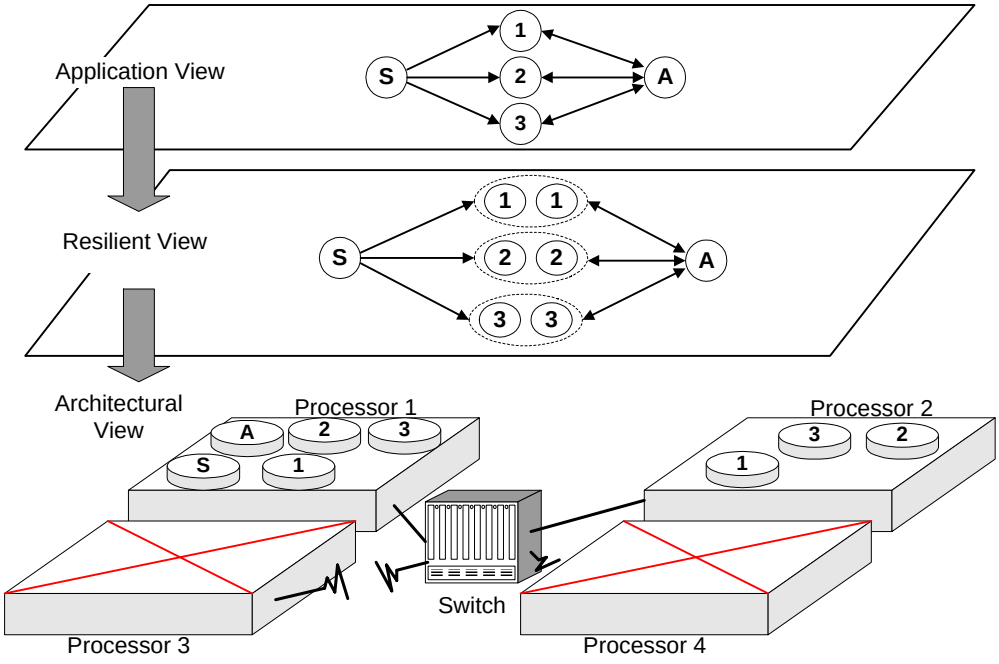


(b)

Figure 4.5: After the First Failure and Recovery



(a)



(b)

Figure 4.6: After the Second Failure and Recovery

4.3 Analytical Model

This section describes the computational requirements of the concurrent sonar processing algorithm in terms of its algorithm characteristics, application dependent properties, and various level of computational resiliency. A predictive model for runtime performance is developed based on these concepts. We also discuss the experimental results to assess the associated overhead of computational resiliency. The model indicates how the changes in application and resiliency parameters affect the total execution time. The motivation in building a performance model is to assess the impact of changes in technology and problem size associated with different applications, allowing cost-performance tradeoffs to be assessed. An analytical model that models the behavior of concurrent computation, sequential computation, and communication overhead based on weighting factors is developed. A linear equation is used to describe the gross behavior of the algorithm executed on a network of shared-memory multi-processors.

The basic notations used in parallel performance measurement are *speedup (sp)* and *efficiency (e)* [Pardalos 1992]. Speedup is a measure that captures the relative benefit of solving a problem in parallel and is defined as the ratio of the time taken to solve a problem on a single processor to the time required to solve the same problem on a parallel computer with P identical processors. Speedup can be defined as

$$sp = \frac{T_s}{\frac{T_s + T_o}{P}}$$

or

$$sp = \frac{T_s P}{T_s + T_o}$$

where T_s is the sequential run time and T_o represents the sum of the overhead that includes communication, idling, or work that is not performed by a sequential algorithm. Only an ideal parallel system can deliver a speedup equal to *P*. In practice, ideal speedup is not achieved because while executing a parallel algorithm, the processors cannot devote 100 percent of their time to computations of the algorithm [Kumar 1994b]. Efficiency is a measure of the fraction of time for which a processor is usually employed and is defined as the ratio of speedup to the number of processors.

$$e = \frac{sp}{P} = \frac{1}{1 + \frac{T_o}{T_s}}$$

Another common observation regarding parallel processing is that every algorithm has a sequential component that will eventually limit the speedup that can be achieved on a parallel computer. *Amdahl's law* indicates: if the sequential component of an algorithm accounts for 1/*s* of the program's execution time, then the maximum possible speedup that can be achieved on a parallel computer is *s* [Amdahl 1967].

The total time for concurrent execution in each processor, T_{conc}, is the sum of computation time, communication costs, and idle time in each processor.

$$T_{conc} = T_{comp} + T_{comm} + T_{idle}$$

The average computation required in each processor, T_{comp}, is equal to the time used to solve the problem sequentially, T_s, divided by number of processors in the system, *P*. A sequential program to be parallelized has two components, the part that can be

parallelized and the part that cannot. Therefore, T_s consists of two times for each component, T_{par}, the time for the part to be parallelized, and T_{seq}, the time for the part to remain sequential.

$$T_s = T_{par} + T_{seq}$$

Hence, the average computation time required in each processor is:

$$T_{comp} = \frac{T_s}{P} \approx \frac{T_{par}}{P} + T_{seq}$$

Idle time occurs only in the fastest computers and can be further avoided by overlapping communication and computation. The total execution time, T_{tot}, can then be defined as the sum of computation and communication time of the slowest processor, ignoring T_{idle}.

$$T_{tot} = T_{comp} + T_{comm} = \frac{T_{par}}{P} + T_{seq} + T_{comm}$$

In this application T_{seq} can be ignored since the ratio of T_{seq} to T_{par} is small, which was 2% in our experimentation. Then, the efficiency of the algorithm can be modeled as follows:

$$e = \frac{sp}{P} = \frac{T_s}{T_{tot} P} = \frac{T_{comp}}{T_{tot}} = \frac{T_{comp}}{T_{comp} + T_{comm}} = \frac{1}{1 + \frac{T_{comm}}{T_{comp}}}$$

We extend this formula to express computational resiliency in the following subsections.

4.3.1 Communication Model

The cost of sending a message between two processors can be represented by two parameters: the message startup time and the transfer time [Foster 1994]. The message startup time, T_{start}, is the time to initiate the communication. Usually, this includes the

time spent in the communication library, system call overhead, and the time for the hardware to begin transmitting [Clement 1993]. The transfer time, T_{trans}, is the time for a message to travel from source to destination across the network, determined by the physical bandwidth of the communication channel. The former cost depends on the speed of the communication hardware and software of each processor. The latter cost depends on how processors are connected. In our experiments we are primarily interested in low-cost, high-performance local area networks based on switched-Ethernet, 100BaseT, and Gigabit. The communication time, T_{comm}, can be modeled as follows:

$$T_{comm} = T_{start} + T_{trans}$$

where T_{start} is the message initialization time and T_{trans} is the transport time.

The transport time is the product of message size (in bytes) and network throughput, T_w, (transport time per byte). Depending on the type of interconnection network and the topology, T_{start} and T_{trans} can be specified differently. In our experiments, modern high-performance network switches were used to connect multiprocessors. With this technology, several multiprocessors can send and receive messages without compromising the network throughput. Thus, assuming the total data of size N is to be divided evenly among P Processors, the communication can be described in the following equation:

$$T_{comm} = T_{start} + T_W \frac{N}{P}$$

Above equation can be extended to express resiliency. Depending on how resiliency is applied in the concurrent sonar processing application, we have two different representations:

(i) If all the threads have resiliency *r*, then communication messages have to be sent from the all the replicas in the source group to all the replicas in the destination group. Therefore, the communication is:

$$T_{comm} = T_{start} + T_W \frac{N}{P} r^2$$

(ii) If only the beamformer threads are replicated with degree of *r* is used:

$$T_{comm} = T_{start} + T_W \frac{N}{P} r$$

In our experiment, the sensor and analysis threads are not replicated while the sub-beamformer threads are replicated, thus we use the formula (ii) to represent the communication time.

4.3.2 Computation Model

To develop the computation model we need to be able to determine the computational complexity of each step in a concurrent algorithm. The complexity of a step is taken to be the time used to complete the step as a function of the problem size [Cormen 1990] and is expressed using weighting factors that represent the relative importance of each step. Recall that the computation time, T_{comp}, is defined as

$$T_{comp} = \frac{T_s}{P}$$

With computational resiliency, it is multiplied by resiliency *r* since the computing time will be increased by the fold of resiliency. Therefore, the new equation is:

$$T_{comp} = \frac{T_s}{P} r$$

In concurrent sonar processing, a beamformer is decomposed into a set of sub-beamformers. Let T_b be the computation time in each sub-beamformer and *k* the number of sub-beamformers, then $T_{par} = kT_b$. For a sonar with *n* sensors, buffer size *m* for a sub-beamformer, and *d* angular resolution, each component of the algorithm can be analyzed as follows where C_i represents the weighting factors:

1. **Sonar Input Signal Generation:** Sonar input signal is calculated for *n* sonar elements. The time required, T_1, is:

$$T_1 = C_1 n$$

2. **Covariance Matrix:** The cost of covariance matrix computation takes *m* multiplications for a $n \times m$ matrix. Thus, the total computation can be defined as follows:

$$T_2 = C_2 n^2 m$$

3. **Beamforming:** This step involves the n^2 multiplications for *d* bearings. Thus, the time required is:

$$T_3 = C_3 d n^2$$

4. **Analyze and Triangulation:** The outputs of beamforming operation is analyzed for *d* angular resolutions. The time required, T_4, is:

$$T_4 = C_4 kd$$

The total time to compute one sonar return in parallel, T_{par}, is thus $T_2 + T_3$. The total time for sequential computation, T_{seq}, is $T_1 + T_4$. The total execution time for a sonar with *n* sensors, *m* buffer size, *kd* angular resolution, *r* resiliency, and *p* processors can then be defined as:

$$T_{tot} = T_{comp} + T_{comm} = \frac{T_{par}}{P} r + T_{seq} + T_{start} + T_w \frac{n^2}{P} r$$

The performance model can thus be described as:

$$T_{tot} = \frac{rk}{P}(C_2 n^2 m + C_3 dn^2) + C_1 n + C_4 kd + C_5 T_W \frac{n^2}{P} r + T_{start}$$

The parallel efficiency can also be predicted with:

$$e = \frac{1}{1 + T_{comm} / T_{comp}} = \left[1 + \frac{T_{start} + C_5 T_w n^2 r / P}{\frac{r}{P}\left(C_2 n^2 m + C_3 dn^2\right)} \right]^{-1}$$

4.3.3 Model Parameters

We have presented the performance model of the sequential algorithm using application dependent parameters and resiliency parameters. Next step of analytical modeling is to decide the weighting factors C_1 through C_5. We utilize linear regression and the least-square fitting method for that purpose [Anton 1994, Noble 1988, Nicholson 1986]. The

least-squares method establishes the qualitative relationship between multiple input variables and one output variable.

The general linear model is expressed as $y = a_0 + a_1 x_1 + a_2 x_2 + ... + a_m x_m$ where a_i, $0 \le i \le m$, is the regression parameter, x_j, $1 \le j \le m$, is the known constants, and y is the output of the linear equation. With n experimental data, the equation can be represented in matrix:

$$y = Xa,$$

$$\text{where } y = \begin{bmatrix} y_1 \\ y_2 \\ : \\ y_n \end{bmatrix}, X = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{1m} \\ 1 & x_{21} & x_{22} & x_{2m} \\ : & : & : & : \\ 1 & x_{n1} & x_{n2} & x_{nm} \end{bmatrix}, a = \begin{bmatrix} a_0 \\ a_1 \\ : \\ a_m \end{bmatrix}$$

We want to choose a so that $a_0 + a_1 x_{i1} + a_2 x_{i2} + ... + a_m x_{im}$ is close to y_i for all $0 \le i \le n$. The residual error for ith data set, r_i, is then represented as

$$r_i = y_i - (a_0 + a_1 x_{i1} + a_2 x_{i2} + ... + a_m x_{im})$$

and the overall error in reproducing the data by the sum of the squares of the residuals is

$$\sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} \{y_i - (a_0 + a_1 x_{i1} + a_2 x_{i2} + ... + a_m x_{im})\}^2$$

The solution that minimizes the overall error can be given by following equation [Anton 1994, Nicholson 1986] :

$$X^T X a = X^T y$$

or

$$a = (X^T X)^{-1} X^T y$$

In our model $m = 5$ represents the number of weighting factors and $n = 15$ is the number of experiments executed to resolve these factors. In our experiments, the threads in this program were partitioned to execute on up to 64 processors. The architecture was organized as 32 Pentium 400 MHz dual-processor computers running the Linux operating system, and connected with 100BaseT networking technology. It forms the isolated homogeneous testbed where all the computers have the same characteristics and capability. The sensor and display were mapped to one machine, while each of the remaining machines executed beamformers. Resiliency was applied uniformly to harden the application by replicating the beamforming elements. Network throughput was measured with Netperf network performance measuring tool [Netperf 1995]. The time used to transfer one byte through the network, T_w, was measured at 0.1 microsecond. The obtained weighting factors are:

T_{start}=1.0e-9, C_1=1.394979e-2, C_2=1.168368e-6, C_3=1.0e-9, C_4=1.0e-9, C_5=3.587773e+1

4.4 Experimental Results

In this sub-section we study the algorithm's scaling properties for all the primary variations of interest, comparing measured and predicted performance results. Figures 4.7, 4.9, and 4.10 show representative experimental results from a broad set of experiments that we have conducted to measure the overhead caused by resiliency and liveness checking. The beamformer was executed once for Figures 4.7 and 4.9, and 100

iterations for Figure 4.10. Each iteration processed a single set of buffered returns. Three parameters were varied in the experiments: the number of processors (1 to 64), the level of replication (1, 2, 3, or 7), and the frequency of the liveness checking (0 to 20 checks over the course of the 100 iterations). The number of sonar elements and the number of buffered returns were fixed to 382 and 2000 respectively.

4.4.1 Scalability

Although resiliency rather than scalability of the concurrent algorithms is the subject of this thesis, it is valuable to ensure that the use of resiliency does not dramatically impact scaling properties. Figure 4.7 shows the scalability of the concurrent algorithm by measuring the execution time of a single beamformer operation with respect to the varying number of processors and differing levels of replication. Uniform decomposition was used, i.e. the number of partitions is equal to that of processors. As with all applications, eventually the effects of diminished granularity outweigh the performance improvement associated with concurrency; each component of work is reduced to the point where the cost of communication dominates. For this particular application and decomposition, the algorithm does not scale linearly after 16 processors. Beamformer threads suffer from communication overhead at this point, for example, the ratio of communication to total execution time increases from 4% with 8 processors to 54% with 64 processors. Predicted execution time for 128 processors shows that the application does not benefit from concurrent processing even with no resiliency.

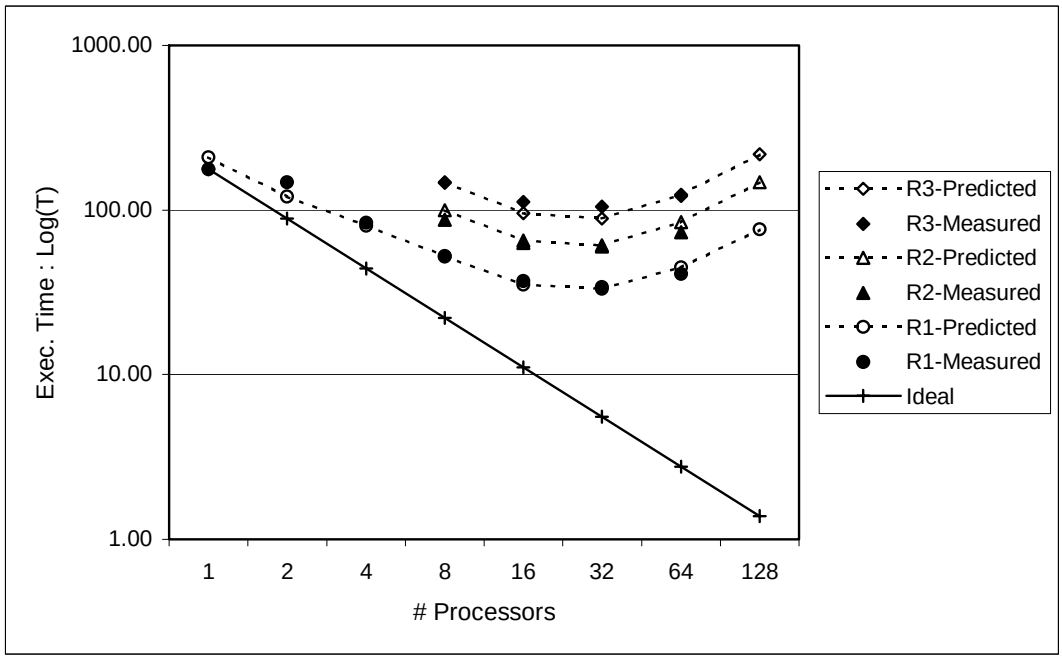


Figure 4.7: Scalability of Concurrent Sonar Processing

4.4.2 Variation in Network Performance

Figure 4.8 shows the predicted execution times with gigabit networking technology as a function of number of processors and the resiliency. From the Figure 4.7 and its analysis, communication overhead was the major bottleneck for the scalability. Using our predictive performance model, we can estimate the performance of the application on different networking technology to see the impact of reduced communication overhead. Time used to transfer one byte through the gigabit network we tested was measured at 0.02 microsecond, approximately four times faster than 100BT network. Predicted results shows that the use of gigabit networking technology increases the scalability of the application further up to 64 processors, but after that point its scalability doesn't improve.

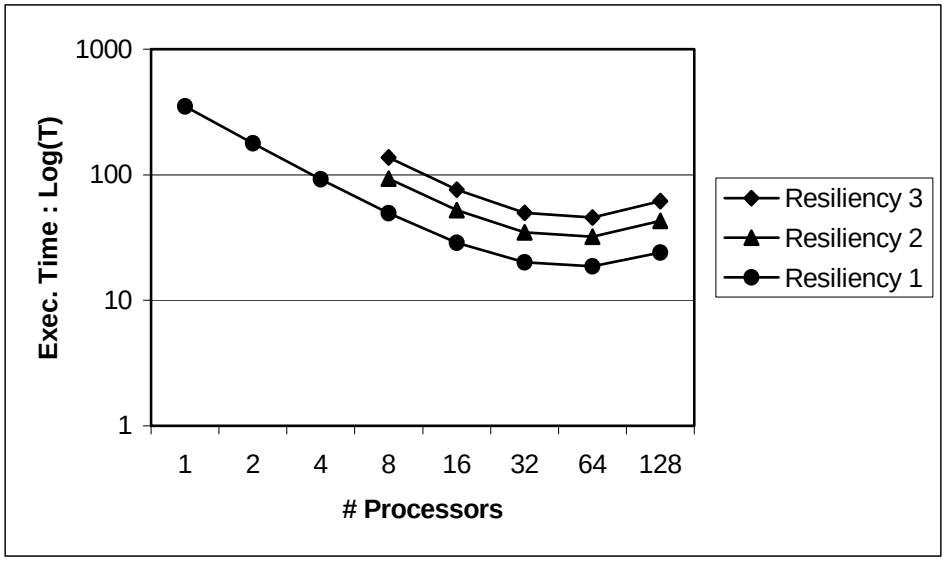


Figure 4.8: Predicted Performance for Gigabit Network

4.4.3 Variation in Resiliency

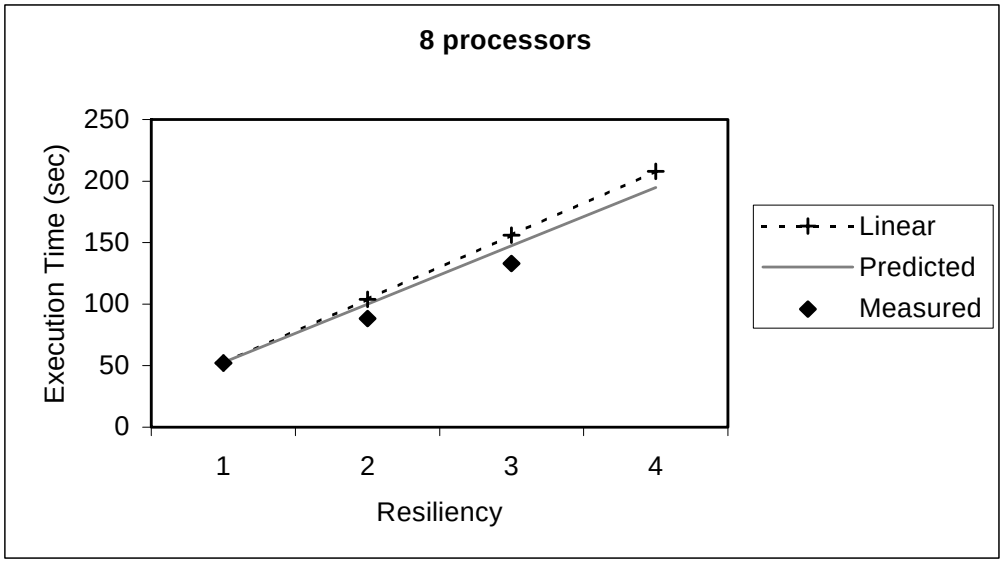
Figure 4.9 shows the overhead of resiliency with respect to the number of processors (8, 16, 32, and 64) in terms of measured time, predicted time, and linear time. Our expectation was that since replication of a thread doubles its computational requirements, level 2 and level 3 resiliency would execute with a two or three-fold decrease in speed respectively. The results indicate however, that in fact performance did not decrease linearly with the level of replication and was less than expected for all the decompositions. The execution time of resiliency 2 increased only 78% over resiliency 1. For resiliency 3, it was as much as 175%.

This artifact resulted from the overlapping of communication and computation in the resilient application: Idle time in the application allowed cycles to be used in completing replicated tasks that would have otherwise been wasted. Obviously, this phenomenon is

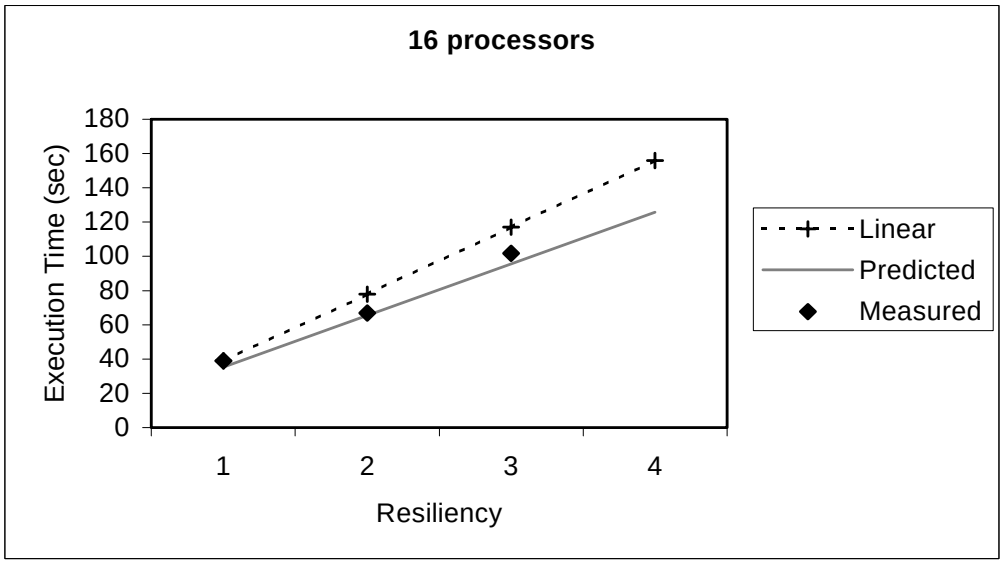
highly application dependent, however, idle cycles can occur for many reasons in distributed applications, e.g. file I/O, synchronization, global operations, etc. Therefore it is not unreasonable to assume that resiliency may often be achievable without significant computational costs.

By studying the experimental results and the analytical model, we can choose the resiliency level for the allowed range of execution time of the application. For example, using 4 processors and no resiliency, the execution time is 84 seconds, but we can achieve the less execution time, 73 seconds, with 64 processors and resiliency 4. Therefore, given the range of the desired response time of the sonar application, we can choose the appropriate resiliency level and the number of processors needed to achieve that. With use of predictive model, we can choose the required number of processors to achieve the allowed response time, if we absolutely need higher reliability. Or, when the number of available processors are limited, we can choose the appropriate level of reliability.

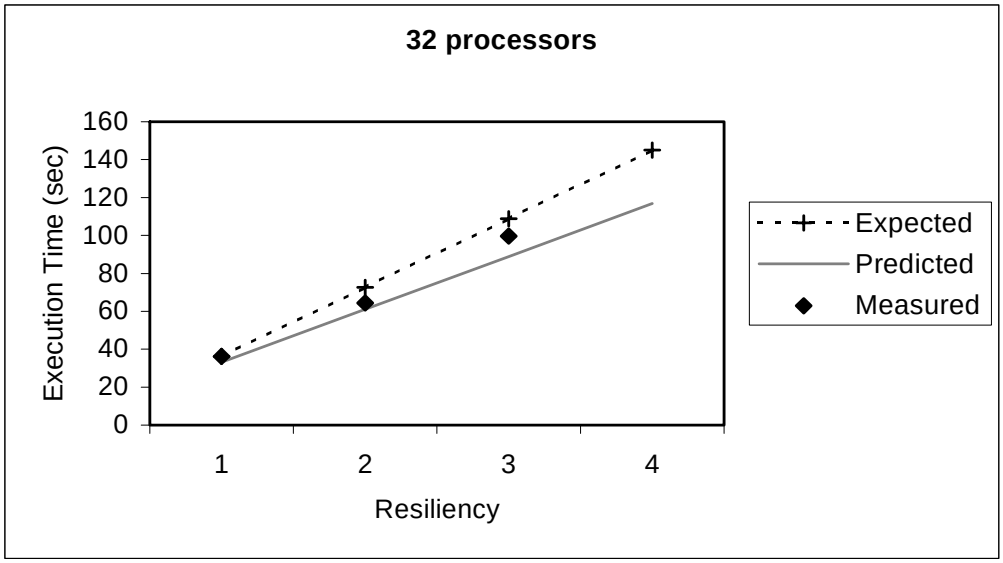
The graph also shows that the plots demonstrate that the accuracy of the predictive model is within 7.2 % for the overall problem size.


8 processors
Execution Time (sec)
250
200
150
100
50
0
1
2
3
4
Resiliency
Linear
Predicted
Measured

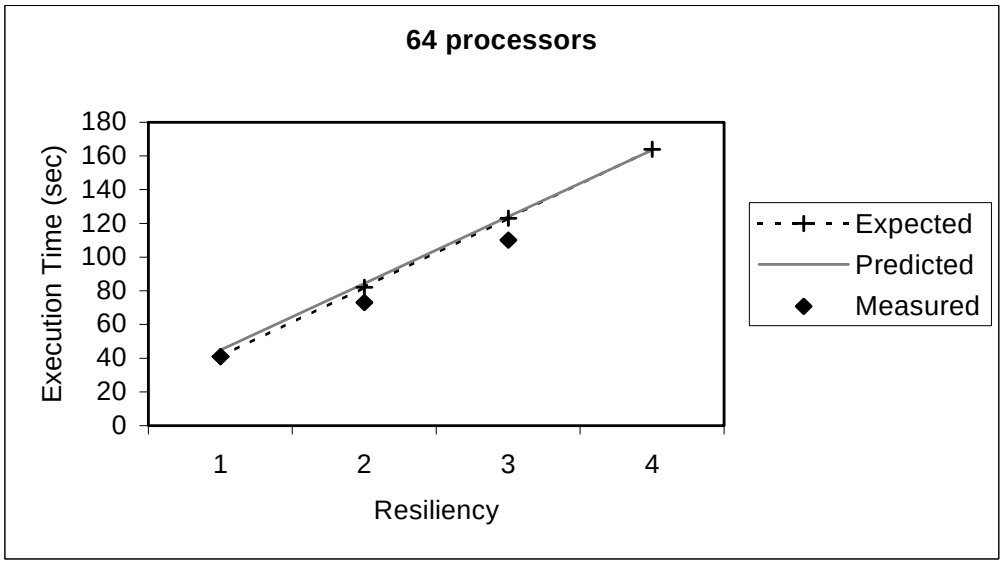
(a)


16 processors
Execution Time (sec)
180
160
140
120
100
80
60
40
20
0
1
2
3
4
Resiliency
Linear
Predicted
Measured

(b)



(c)



(d)

Figure 4.9: Overhead of Resiliency

4.4.4 Variation in Frequency of Liveness Checking

Example results concerning the frequency of liveness checking are presented in Figure 4.10. For these results, the problem was decomposed in to 32. The lower three lines show the performance of resiliency 1, 2, and 3 using 32 processors. The top line represents resiliency 7. It used more processors, 56, for the same number of decompositions to avoid excessive load of computation per processor. Even though resiliency 7 may seem to be a high level of replication, we consider this case interesting to since it more closely approximates the computational model presented in chapter 3. These results show that the use of liveness checks does not incur noticeable overhead for all cases. The overheads never exceeded 1% even when liveness checking is frequent (once every 5 iterations of the beamformer) and the level of resiliency is high, i.e. 7. In addition, the overhead of resiliency for level 7 was only 410 % over resiliency 1, indicating that very high levels of survivability may be possible without a direct linear cost.

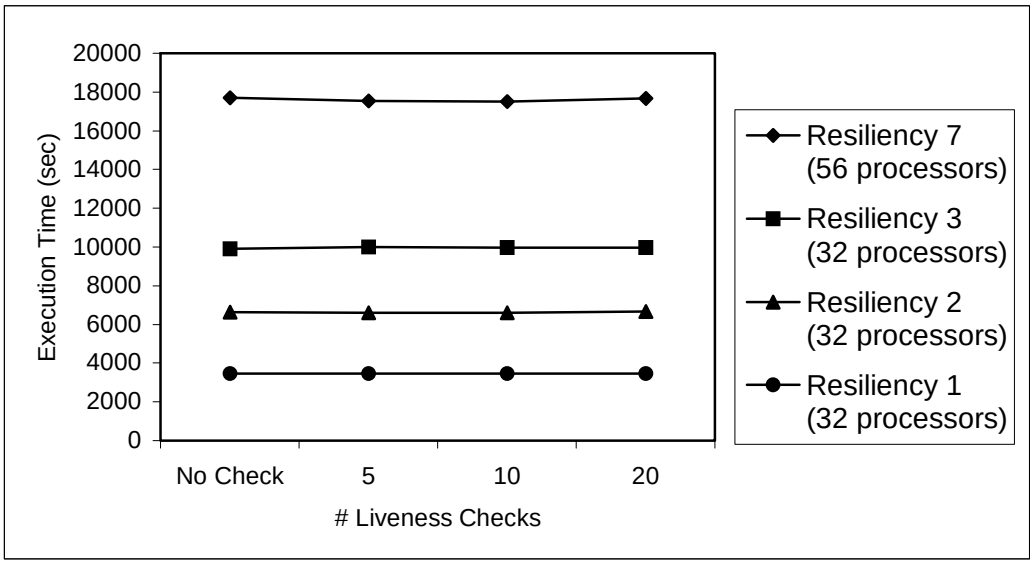


Figure 4.10: Overhead of Liveness Checking

We have also measured the recovery overhead during the liveness checking in the presence of failure. Time required to recover from the failure and recreate a new thread consists of times to create a new thread at another location, to transfer some system information, and to transfer user specified data structures. The first two components are common overhead regardless of the applications while the third component may vary depending on the applications. We measured the amount of time needed for the first two components, which was 3 *ms* in our experimentation environment.

4.5 Summary

This chapter has developed concurrent sonar application that uses computation resiliency and the associated performance models. The experimental results revealed the associated overhead due to resiliency and the frequency of liveness checking. The model was validated against experimental data on homogeneous distributed computing environments where all the processors are identical. Using this model, a wide range of practical design questions can be answered. For example:

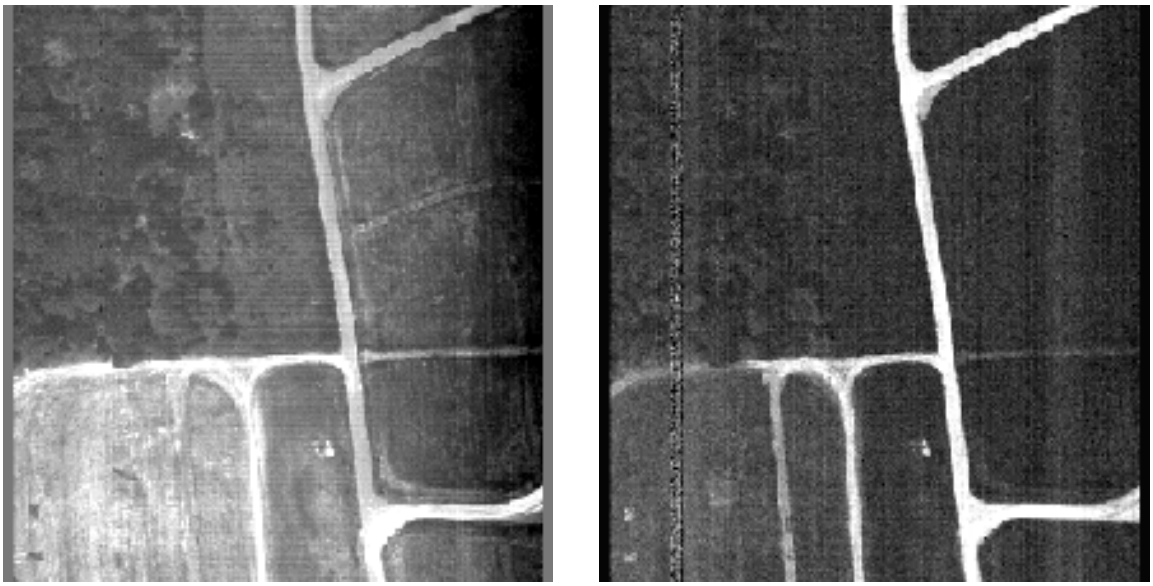
- For a given fixed cost, what performance and reliability can be expected from the applications?
- What level of resiliency can be achieved without compromising the performance of the application?
- How often a liveness checking can be performed to assure the level of system operability?
- What network speed will realize my cost-performance objectives?

Chapter 5 Remote Sensing Application

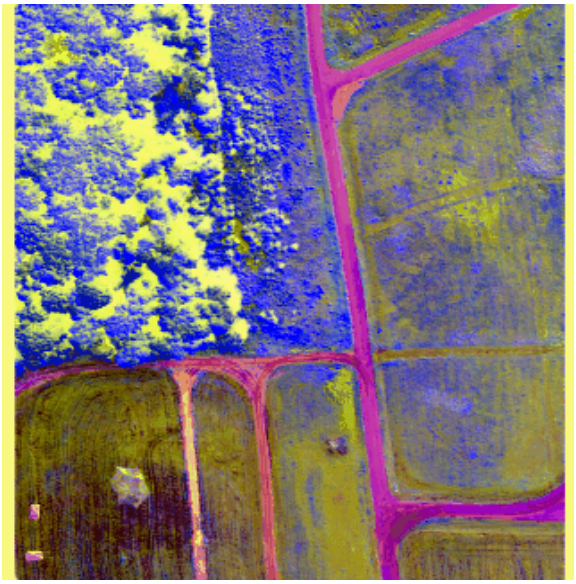
This chapter presents another application, remote sensing application, to which computational resiliency can be applied. Experimental studies and an analytical model for this applications are also presented. The same methods of experimentation and analysis in chater 4 are used.

5.1 Introduction

A second application to which we have applied resiliency is a concurrent *spectral-screening PCT algorithm* (s-PCT) that can be used for remote sensing applications [Achalakul 2000]. The algorithm takes as input a large number of grey-scale images emanating from a hyper-spectral sensor. Each image corresponds to a particular wavelength of light, for example, Figure 5.1a shows the image taken at 1998nm using a 210-channel hyper-spectral image collected with the Hyper-spectral Digital Imagery Collection Experiment (HYDICE) sensor, an airborne imaging spectrometer. The HYDICE image set corresponds to foliated scenes taken from an altitude of 2000 to 7500 meters at wavelengths between 400nm and 2.5 micron. The scenes contain mechanized vehicles sitting in open fields as well as under camouflage. The s-PCT algorithm removes redundancy in the image set and presents a single color composite image that shows the important spectral contrast. For example, Figure 5.1b shows the output of the algorithm in which the mechanized vehicles are clearly visible in the lower left of the figure due to spectral contrast.



(a) 400 and 1998 nm



(b) Color-Composite Image

Figure 5.1: Concurrent Remote Sensing

5.2 Computational Resiliency

The distributed version of the s-PCT algorithm uses the standard manager/worker decomposition technique [Chandy 1992] as shown in Figure 5.2. A sensor thread generates and partitions the 210-frame image cube into sub-cubes and distributes the sub-cubes to worker threads. A manager synchronizes the actions of these workers, accumulates partial results, and displays the resulting image.

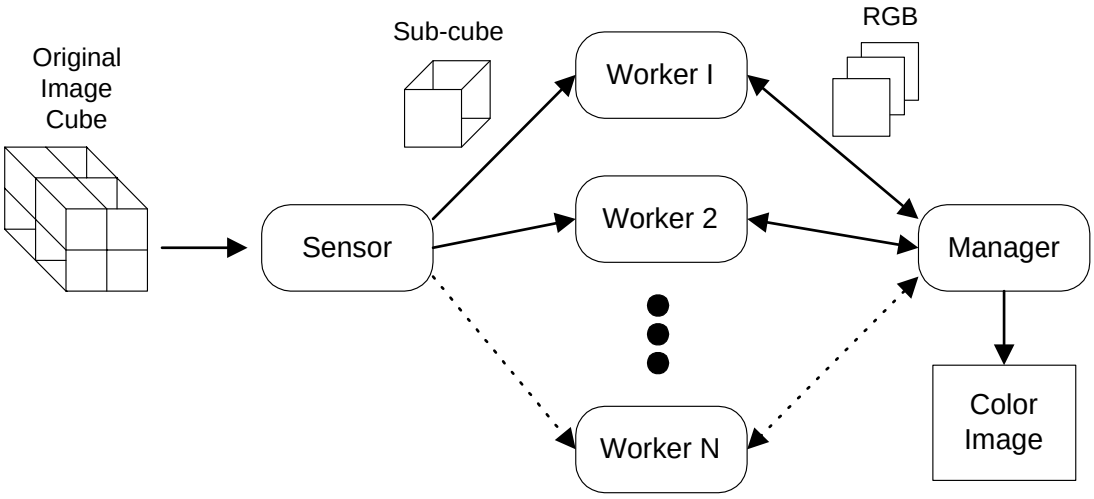


Figure 5.2: Manager/Worker Communication Model

Figure 5.3 shows the application view without resiliency. There are six threads corresponding to the sensor, manager, and workers connected through communication channels.

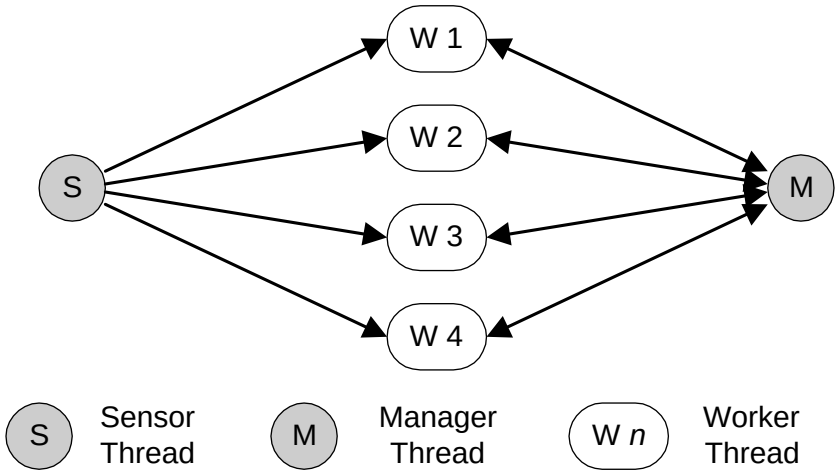


Figure 5.3: Application View

Figure 5.4 shows the resilient view of the same application in computational resiliency. In this picture, worker threads are replicated with degree of three.

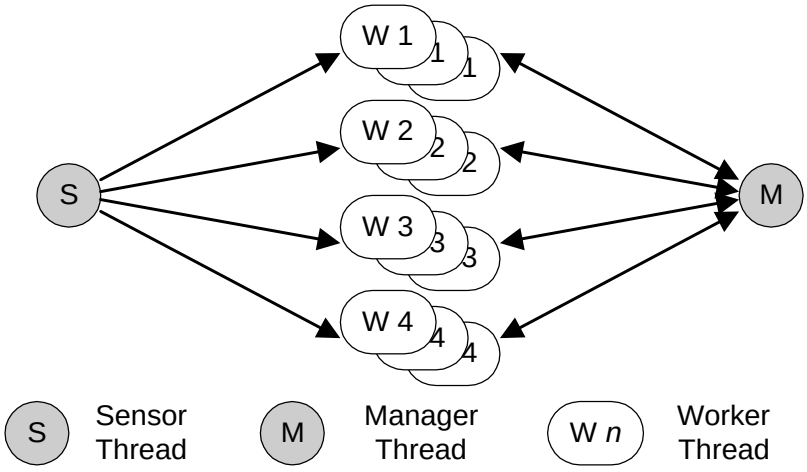


Figure 5.4: Resilient View

Program 5.1 shows the abstract code for initializing remote sensing application. The user specifies the required resiliency, one for both sensor (1) and manager (2) threads and

three for each worker thread (3). Each worker is associated with a communication channel to the sensor (4) and the manager thread (5).

```
remote_sensing () {
  gs = group_create (sensor_fn, 1) @ C0                          // 1
  gm = group_create (manager_fn, 1) @ C0                         // 2
  for (i=1; i<=4; i++) {
    gw[i] = group_create (worker_fn, 3) @ Ci, Ci+1, Ci+2         // 3
    channel_create (gs, gw[i])                                   // 4
    channel_create (gw[i], ga)                                   // 5
  }
}
```

Program 5.1: Abstract Code for Initializing Remote Sensing Application

Each worker thread executes the algorithm shown in Program 5.2 and maintains a set of sub-cubes (1,4) to operate on. An initial request is sent to the sensor to obtain the first sub-cube (2). After this initial request, the processing of each sub-cube is overlapped with communication of the remaining the next sub-cube from the sensor (3). This represents the primary communication step in the algorithm and corresponds to distributing $1/n^{th}$ of the image cube to each of n worker threads.

The spectral screening algorithm produces a set of unique spectra. Although each sub-cube contributes to this set through an appropriate abstract operation (6), the set must be accumulated across all sub-cubes. This accumulation is performed through communication with the manager. Each worker sends a prospective subset of the spectra to the manager (7) and overlaps this communication with computation of the next subset. When all sub-cubes have been processed, the manger transmits the resulting unique set to

all workers (8). Typically, the amount of communication in this step is orders of magnitude less than the size of an image cube.

When the spectral screening is completed globally, the algorithm proceeds to compute a set of statistics (mean-vector and covariant-sum) that give a measure of the variation in images at each spectra. Although, once again, the statistics can be largely computed on a per sub-cube basis using an appropriate abstract operation (9), the manager is again involved in assembling the statistics to form a transformation matrix A and mean-vector m (10,11). The communication involved in this step is on the order of n^2 where n is the number of spectra, again typically significantly smaller than the size of the image cube.

With the matrix A and mean-vector m available, a PCT (12) and human-centered mapping (13) can be computed on each sub-cube independently to produce a patch of the final color image. The patches are accumulated at the manager for display (14). Thus, the final communication is only m^2, where m is the size of the image. Periodic liveness checking is performed when appropriate (15).

```
worker_fn() {
cubes = {}                                                    // 1
group_send(request,sensor)                                    // 2
while(numsubcubes <= numcubes/numworkers) {
        subcube = recv(sensor)                                // 3
        cubes = cubes U subcube                               // 4
        group_send(request,sensor)                            // 5
        ssubset = spectral_screening(subcube)                 // 6
        group_send(ssubset, manager)                          // 7
}
sset = group_recv(manager)                                    // 8
substats = statistics(sset)                                   // 9
group_send(manager, substats)                                 // 10
[A, m] = group_recv(manager)                                  // 11
subcomponents = PCT(A, m, cubes)                              // 12
subimage = human_centered_mapping(subcomponents)              // 13
group_send(subimage, manager)                                 // 14
    if (time expires)
       liveness_check ()                                      // 15
   }
}
```

Program 5.2: Abstract Code for Worker Thread

5.3 Analytical Model

We apply the same linear regression and least-squares method to develop the analytical model for remote sensing application. The predictive model describes the performance of the distributed algorithm in terms of the number of spectra, the image size, network bandwidth, the number of processors, and the replication level. The model is represented as a linear equation of terms for each step of the algorithm with weighting factors. We calibrate the weighting factors from the actual experimental data.

5.3.1 Communication Model

Recall that the communication cost can be modeled as

$$T_{comm} = T_{start} + T_w \frac{N}{P} r$$

since the sensor and manager threads are not replicated.

5.3.2 Computation Model

Recall that the computation time, T_{comp}, is defined as

$$T_{comp} = \frac{T_{par}}{P} r + T_{seq}$$

In the concurrent algorithm the original hyper- or multi-spectral image cube is decomposed into a set of sub-cubes where each sub-cube is distributed to a worker. The parallel time, T_{par}, can then be defined as follows:

$$T_{par} = kT_b$$

where k is the number of sub-cubes and T_b is the time used to compute one sub-cube.

Let m be the width and height of each sub-cube in the hyper-spectral image, n be the number of spectral band, s be the number of unique spectra per sub-cube, and p be the number of processors. Considering each component of the algorithm in turn:

1. **Spectral screening:** The computation associated with this step involves a calculation taken over all pixel vectors concurrently, m^2 at each worker. Each computation (the

arccosine of the dotproduct of the pixel vectors pair) involves the calculation between a new vector (of size *n*) and all vectors in the unique set (*s*). Thus the time required, T_1, is:

$$T_1 = C_1 m^2 sn$$

2. **Merge unique sets:** This step is computed sequentially at the manager. The computation involves an angle calculation associated with each pixel vector (of size *n*) in *p-1* sets, where each set contains *s* pixel vectors. The time required, T_2, is:

$$T_2 = C_2 (p-1) sn$$

3. **Mean vector:** This step involves taking an average of the pixel values in a unique spectral set at the manager. The number of operations is related to the number of unique spectra (*s*) and the number of frames (*n*). The time required, T_3, is:

$$T_3 = C_3 sn$$

4. **Covariance sum**: The computation associated with the covariance sum is performed over the pixels in a unique set of size *s* at the worker. Each computation on a pixel involves matrix multiplication (complexity of n^2). The time required, T_4, is:

$$T_4 = C_4 n^2 s$$

5. **Covariance matrix:** This computation involves forming the matrix sum of the matrices returned from the previous steps at the manager. There are *p* matrices of size n*x*n. The time required, T_5, is:

$$T_5 = C_5 n^2 p$$

6. **Transformation matrix:** The time used in this step is dominated by the time used to calculate eigenvectors at the manager. The time required, T_6, is:

$$T_6 = C_6 n^3$$

7. **Transformation of the data:** The computation in this step is performed over the pixels in an image of size m^2. Each computation on a pixel involves matrix multiplication with the complexity of n^2 at the worker. The time required, T_7, is:

$$T_7 = C_7 n^2 m^2$$

8. **Color mapping:** This step of the algorithm involves linear transformation of the first three principal components in achromatic, red-green, and blue-yellow opponency at the worker. The time required, T_8, is directly proportional to the size of the sub-cube:

$$T_8 = C_8 m^2$$

The total time to compute one sub-cube, T_b, is thus $T_1 + T_3 + T_4 + T_7 + T_8$. The total time for sequential computation T_{seq}, is $T_2 + T_5 + T_6$. The total execution time, T_{tot}, for an n-band image cube of size mxmxp, can then be defined as:

$$T_{tot} = T_{comp} + T_{comm} + T_{seq} = \frac{T_{par}}{p} r + T_{seq} + T_{start} + T_w \frac{km^2 n}{p} r$$

$$= \frac{kT_b}{p} r + T_{seq} + T_w \frac{km^2 n}{p} r + T_{start}$$

The performance model can thus be described as:

$$T_{tot} = \frac{kr}{p}(C_1 m^2 sn + C_3 sn + C_4 n^2 s + C_7 n^2 m^2 + C_8 m^2) +$$

$$(C_2(p-1)sn + C_5 n^2 p + C_6 n^3) + C_9 T_w \frac{km^2 n}{p} + T_{start}$$

The parallel efficiency can also be predicted with:

$$e = \frac{1}{1 + T_{comm} / T_{comp}} = \left[1 + \frac{T_{start} + C_9 T_w krm^2 n / p}{\frac{kr}{p}(C_1 m^2 sn + C_3 sn + C_4 n^2 s + C_7 n^2 m^2 + C_8 m^2)} \right]^{-1}$$

5.3.3 Model Parameters

On a 100 baseT network the network throughput, T_w, was measured at 0.1 microsecond. The associated experiments were performed on the same homogeneous testbed as with concurrent sonar processing. After approximately 15 experiments, the value of the weighting factors were obtained, and the final values are listed below:

T_{start} =1.0e-009, C_1 = -7.0523573e-001, C_2 = -1412.1185, C_3 =350.61074,

C_4 =3.2096624e-007, C_5 =1.0e-009, C_6 =1.0e-009, C_7 = 1.3325618e-005

C_8 = -1.8525113e-006, C_9 = 21.119059

5.4 Experimental Results

The performance of the algorithm was measured on the same distributed environment used for concurrent sonar processing in the previous section. The same experiment was conducted with all workers replicated up to the level of seven; the manager and sensor were not replicated.

5.4.1 Scalability

Figure 5.5 shows the speedup gained as a function of the number of processors both with and without resiliency. Once again, as we would expect, eventually granularity concerns begin to reduce the speedup of the algorithm. For example, from 2 processors to 4 processors, the performance improved 97% while from 32 processors to 64 processors only 38% improvement was observed.

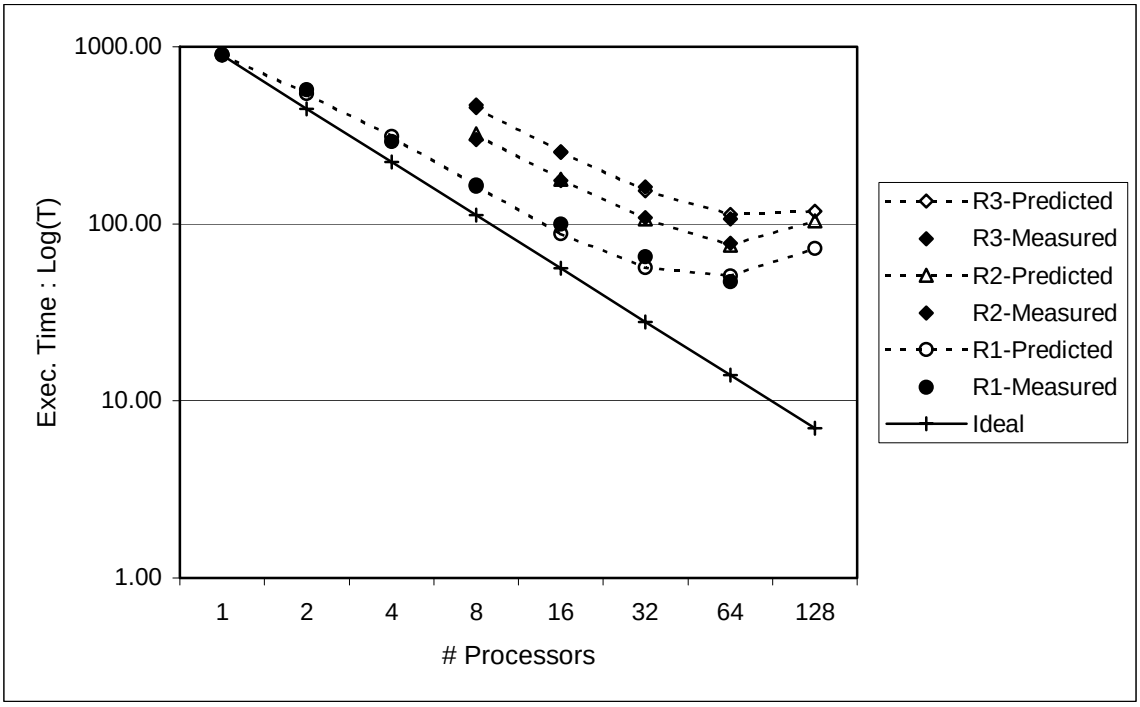


Figure 5.5: Scalability of Concurrent PCT

5.4.2 Variation in Network Performance

Figure 5.6 shows the predicted execution time with gigabit networking technology. Predicted results indicate that the use of faster networking technology can improve the scalability of the application a lot even with higher resiliency.

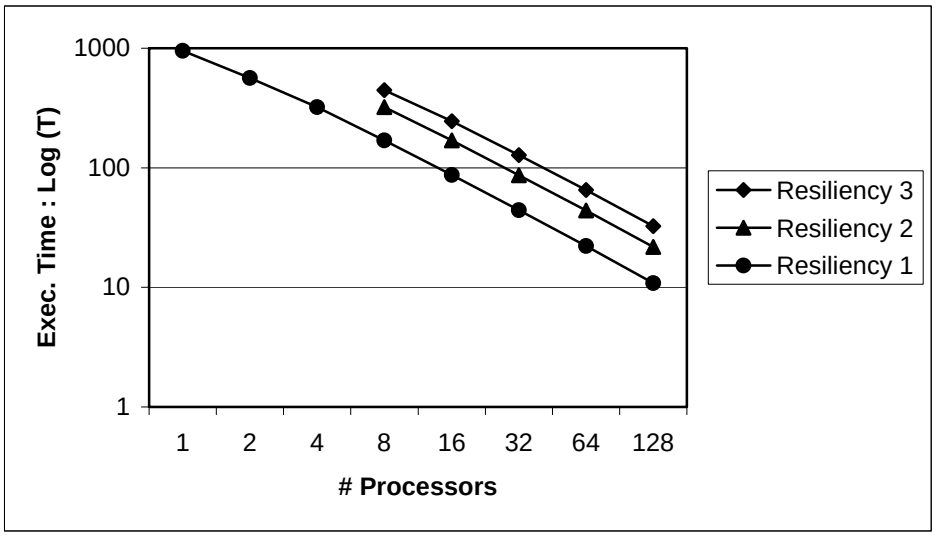
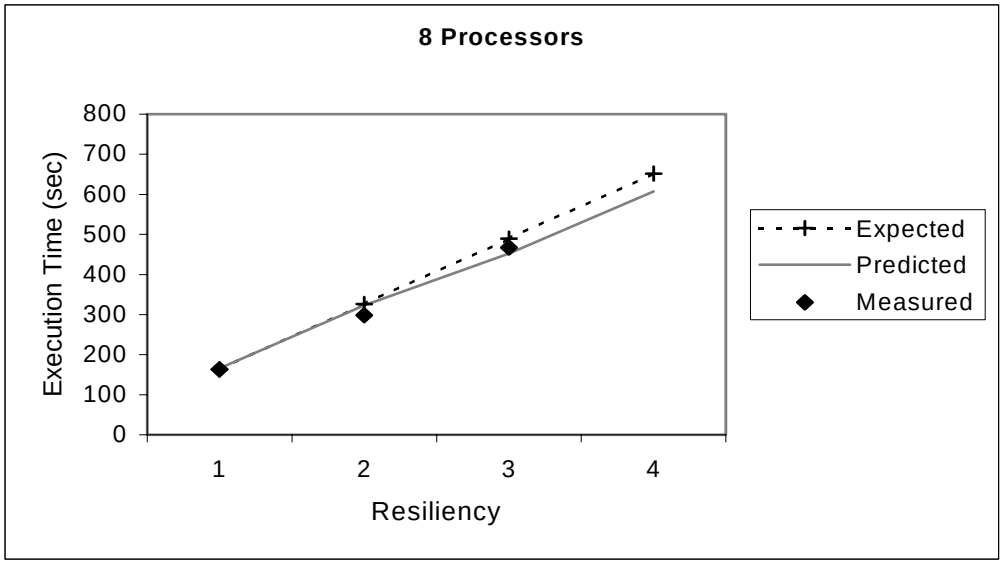


Figure 5.6: Predicted Performance for Gigabit Network

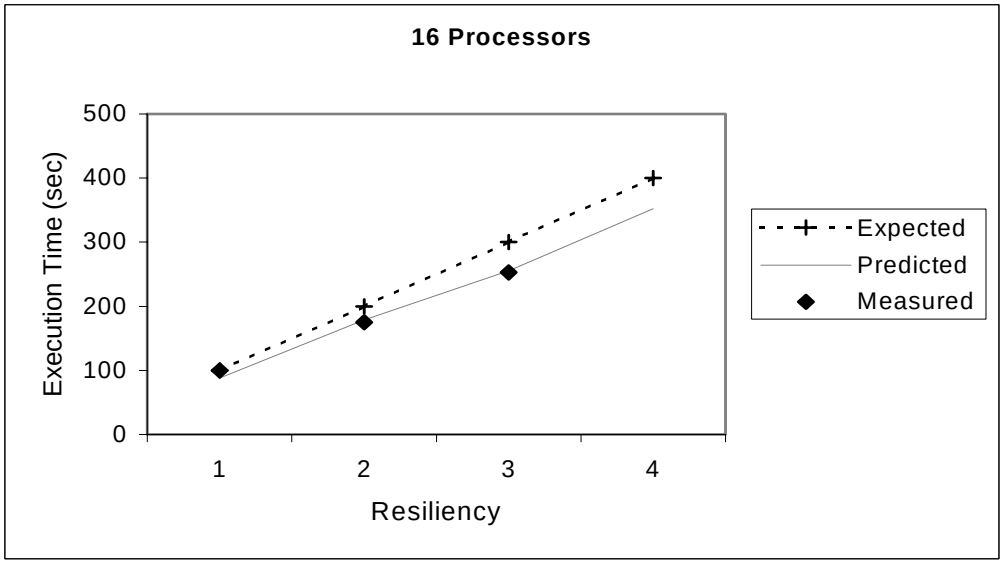
5.4.3 Variation in Resiliency

Once again, when resiliency was applied the expected result was that performance would decrease by a factor of two or three depending on the specified resiliency since the replicated processes require both memory and processor resources. Figure 5.7 shows the overhead of resiliency with respect to the number of processors. Notice that the overhead caused by resiliency 2 is only 82% over resiliency 1, and 186% for resiliency 3 respectively. As in the sonar application, we observe that resiliency is able to utilize idle cycles in the concurrent algorithm to reduce the cost of replication.

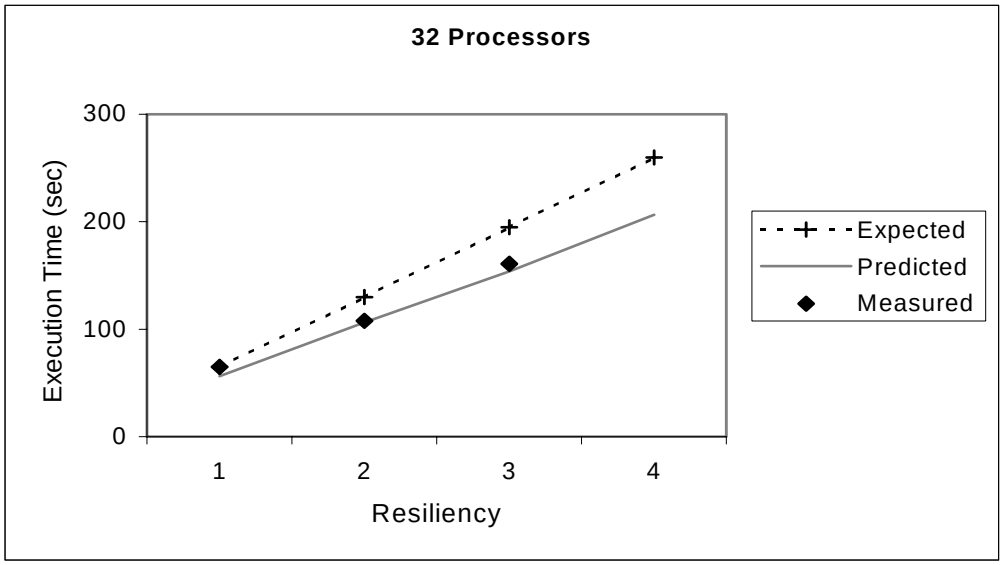
The graph also shows that the plots demonstrate that the accuracy of the predictive model is within 5.4 %.


8 Processors
Execution Time (sec)
800
700
600
500
400
300
200
100
0
1
2
3
4
Resiliency
Expected
Predicted
Measured

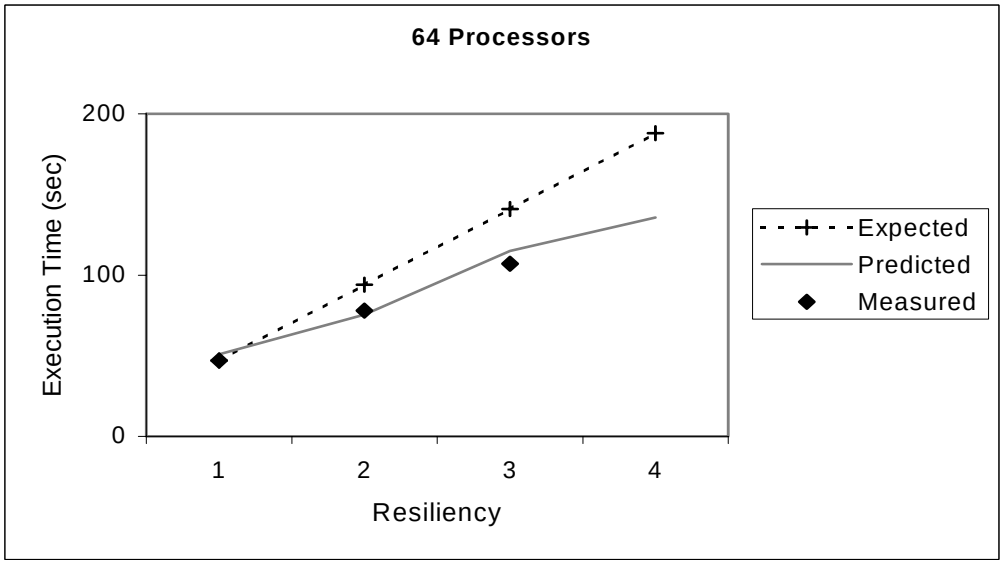
(a)


16 Processors
Execution Time (sec)
500
400
300
200
100
0
1
2
3
4
Resiliency
Expected
Predicted
Measured

(b)



(c)



(d)

Figure 5.7: Overhead of Resiliency

5.4.4 Variation in Frequency of Liveness Checking

Figure 5.8 shows the overhead caused by the liveness checking. In each case, the overhead was less than 1%. It indicates that the resiliency is the primary source of overhead, and the frequent use of liveness checking would not add noticeable. This is beneficial in that frequent use of liveness checking allows an application to recover from the possible failure more quickly. The overhead of replication for resiliency 7 was 414% over resiliency 1, a considerable improvement over the expected factor of 7.

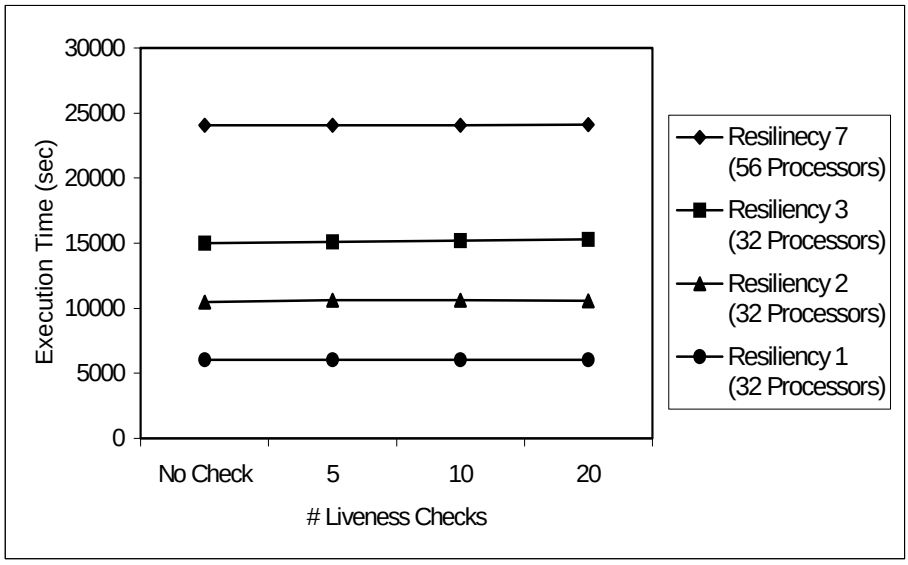


Figure 5.8: Overhead of Liveness Checking

5.5 Summary

This chapter has developed a prototypical application, remote sensing, that can benefit from computation resiliency. The associated predictive analytical model was presented.

Various experimentations were used to validate the model that can be used to make predictions.

Throughout chapter 4 and 5, the thesis showed how the concepts and library can be applied in the context of two realistic military applications: a towed array sonar and a remote sensing application. The implementations of these applications were studied to ascertain the overheads associated with the technology on a moderately scaled, homogeneous architecture consisting of 32 high-performance dual-processor PC's connected with 100Mbit/sec Ethernet technology.

For both applications, ability to utilize idle cycles to reduce the cost of increased survivability was evident, especially at higher levels of redundancy than one normally considers practical. This higher level is directly motivated by the computational model which provides strength in numbers. Although initially, the use of group based liveness checking was considered to be a significant defect with the current implementation strategy, it has proved to be less problematic than expected accounting for less than a 1% overhead in both applications. In both applications, reducing the frequency of checking could have reduced the overhead still further.

Many aspects of computational resiliency remain to be explored and several alternative implementation strategies have yet to be tested. However, the results in this thesis indicate that the general concept is both practical and has less cost than originally anticipated.

Chapter 6 Heterogeneous Systems

Since machines in the networked environment usually have widely different performance and memory characteristics, load balancing techniques are required. These techniques must disperse replicated structures to realize improved reliability. To explore the performance issues associated with heterogeneity of the distributed computing environment, we have incorporated the technology into two prototype distributed applications: a towed array sonar and a hyper-spectral remote sensor. In this chapter we outline the load balancing techniques, and show how they are applied to the applications. Performance measurements are provided that quantify the overhead of resiliency, under normal operating conditions, using a network architecture containing 21 heterogeneous computers connected with both Gigabit and Fast Ethernet technologies.

6.1 Load Balancing Algorithm

Efficient allocation of the replicas in computational resiliency improves the performance and reliability of the application. Traditional load balancing techniques address the optimal allocation of resources to tasks. We augment this process with reliability constraints.

Figure 6.1 shows that load balancing strategy in computational resiliency. Each compute has different capability and the workload of the systems is balanced through allocating the replicas of threads considering the utilization of the computers. For example, three

threads are allocated to processor 1 while only one is allocated to processor 4. Application view and resilient view don't change while the actual mapping of the threads to the system realizes load balancing.

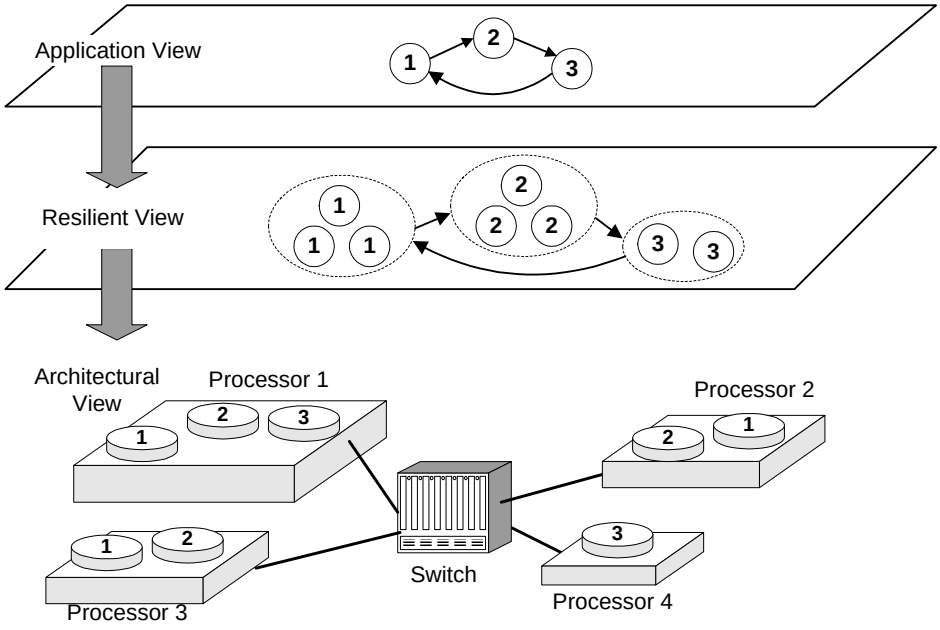


Figure 6.1: Load Balancing in Computational Resiliency

Time complexity of finding out the optimal allocation of the replicated threads over a given set of processors is exponential. Even with moderately sized system, task allocation takes too much time to be practical, which results in increasing the recovery time when failure occurs. In that sense, allocating replicas on-line and in real-time is desirable. Powerfulness of an on-line algorithm is usually represented by competitive ratio, i.e., ratio between the performance achieved by the on-line algorithm and that of off-line algorithm. Azar [Azar 1992a, 1992b] proved that greedy strategy based on-line load balancing technique can achieve a competitive ratio of $\lceil \log_2 n \rceil + 1$.

Program 6.1 outlines the greedy algorithm used, where l_j represents the load of task *j*, L_i the load on processor *i*, r_j the number of replicas for task *j*, T_i a set of tasks mapped to computer *i*, and S_j a set of computers to which task *j* may not be allocated.

```
for all i
    T_i ← φ
end for
for each task j
    S_j ← φ
    while r_j > 0
        choose processor i that is not in S_j and
            has the lowest utilization
        if the reliability constraints are met
            assign the replica of task j to computer i
            L_i = L_i + l_j
            T_i ← T_i ∪ {task j}
            r_j = r_j − 1
        else
            S_j ← S_j ∪ {i}
        end if
    end while
end for
```

Program 6.1: Load Balancing Algorithm

In determining the load of a task, there are two options. One is to use abstract, algorithmic quantities such as the number of operations or data structures. If the load of task *j* is taken to be l_j, then the load of computer *i* is $L_i = \sum_{j \in T_i} l_j$. In that case, the utilization of computer *i* is given, $U_i = \frac{L_i}{C_i}$, where C_i is the computer's capacity.

Similarly, the utilizations due to the tasks are given by $u_j = \frac{l_j}{C_{M(j)}}$, where $M(j)$ is the computer to which task *j* is mapped. As a second option, one can measure the utilization using system facilities to get CPU time or amount of memory used by a task. In that case, one simply reverses the above formulas to calculate the abstract loads: $l_j = C_{M(j)} u_j$ gives the load of a task. The resulting task loads and utilizations are summed to yield the computer's total loads and utilizations, respectively.

In both cases, it is assumed that one knows the resource capacity of a given computer. The capacity can be determined in a number of ways. If the capacity measured is processing speed, a benchmarking program – possibly the target application with a smaller test problem – can be used to determine the relative speeds of various machines. Theses off-line performance numbers, along with other statistics, such as the machines' memory capacities, can be put into a file which is read at the start of the computation. A third measure is to use both invariant algorithmic quantities and system-measured utilization numbers. For examples, one might use the number of iterations in an application as well as the CPU time required to process those iterations. By dividing former by the latter, one can calculate the capacity of the system dynamically during the execution.

To assess the load of a task, we measure the execution time of a standard benchmark task on the slowest computer in the network, and assess the relative performance of any other computers. We assume that faster processors have a larger capacity based on relative

speeds and this allows the algorithm to determine the processor with lowest utilization.

The reliability constraints assure that each replicated task is assigned to a distinct computer since the failure of a computer results in loss of all the replicas in it. In addition, if more than one LAN is in use, loss of network connectivity between LANs may reduce reliability. Thus we ensure that replicas within a group are allocated to different LANs where possible.

6.2 Heterogeneity in Data Representation

Another aspect of heterogeneity is the data representation scheme of each computer architecture. Each computer has different number of bytes to represent the same tyep of data. There is also Endian byte ordering problem[Cohen 1981]. Big Endian orders the bytes by placing the most significant byte first while Little Endian the least significatn byte first. Figure 6.2 shows two data representation mechanisms.

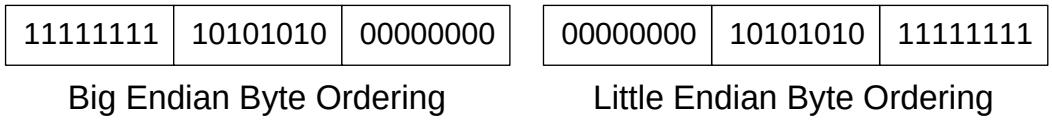


Figure 6.2: Endian Byte Ordeing

Table 6.1 shows the characteritics of each computer architecture.

Processor	**Operating System**	**Byte Ordering**	**Long Integer Bytes**
Intel Pentium III	Windows NT	Little Endian	4
Alpha	LINUX 2.2.1	Little Endian	8
R 4400	IRXI 6.4	Big Endian	4

Table 6.1: Data Representations on Heterogeneous Computer Architectures

These differences in representing data affect the implementation of computational resiliency. All the control and data messages in the system have to be compatible with each other. Control messages used in three protocols in computatil resiliency library and the data messages sent and received at user level have to be understood in each processor. For this purpose, each communication channel should be able to understand different representation of the messages. If the communication channel is between homogeneous systems, no trasformation is applied. Between heterogneous computers, transformation to neural representatoin is enforced. At the reciver side, this neutral format is transformed to local data presentation.

Figure 6.3 shows the impact of heterogeneity in flow control protocol of computational resiliency. Each replica of a thread may be spawned on different types of processors. Then, in flow control protocol, the same message is received by each of those replicas in different processor type, and each message may be in different format. Flow control protocol in heterogeneous environment need to transform these messages to the appripriate format understood locally, discards the duplicates, and deliver them in order to upper level.

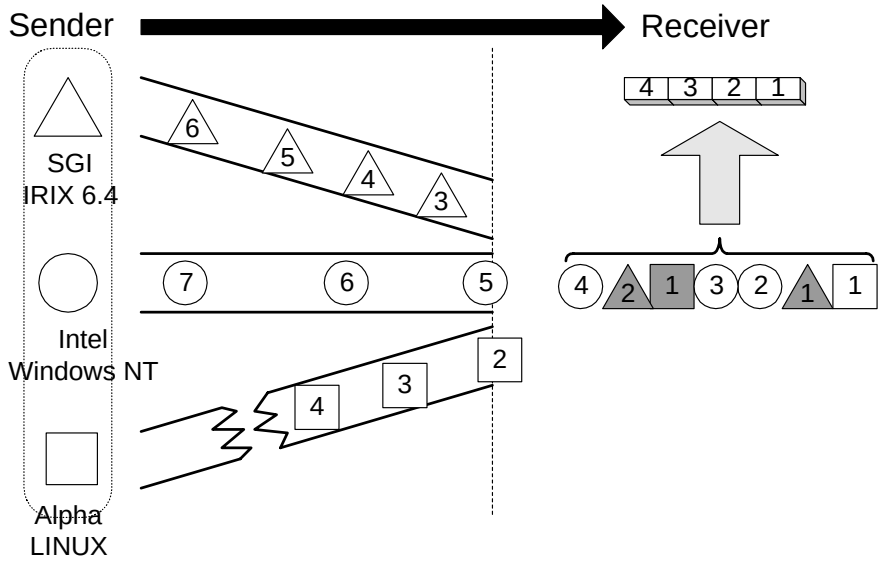


Figure 6.3: Flow Control in Heterogeneous Environments

6.3 Experimental Testebed

To explore the feasibility of these concepts, two prototype applications were developed and mapped to a network architecture organized as 21 heterogeneous computers connected with a 100BT and Gigabit Ethernet switches. These computers included a broad range of performance and memory characteristics, operating systems, and byte orderings. They were:

	Processor Type	Memory Size	Operating System	Network (Mbps)	Relative Speed
0	450 MHz Pentium III (x4)	1.5 GB	Windows NT 4.0 Enterprise Server	1000	9.3 (x4)
1	300 MHz Pentium II (x2)	256 MB	Windows NT 4.0 Server	1000	2.6 (x2)
2	500 MHz Pentium III (x8)	4 GB	Windows NT 4.0 Enterprise Server	1000	14.4 (x8)
3	500 MHz Pentium III (x8)	4 GB	Windows NT 4.0 Enterprise Server	1000	14.4 (x8)
4	500 MHz Pentium III	128 MB	Windows NT 4.0	100	2.6
5	500 MHz Pentium III	128 MB	Windows NT 4.0	100	2.6
6	533 MHz Celeron	128 MB	Windows NT 4.0	100	2.2
7	533 MHz Celeron	128 MB	Windows NT 4.0	100	2.2
8	533 MHz Celeron	128 MB	Windows NT 4.0	100	2.2
9	533 MHz Celeron	128 MB	Windows NT 4.0	100	2.2
10	200 MHz R4400	128 MB	IRIX 6.4	100	1
11	150 MHz R4400	288 MB	IRIX 6.4	100	1.3
12	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)
13	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)
14	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)
15	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)
16	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)
17	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)
18	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)
19	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)
20	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)

Table 6.2: Heterogeneous Computers Characteristics

The performance of each of these machines was measured relative to the slowest machine, the 200 MHz Indigo II, and is shown in brackets. Note that the performance of the machines varied by a factor of almost 14.4(x8), and the available memory varies by a factor of 32. Figure 6.4 shows the overall networking structure composed of both Gigabit Ethernet and Fast Ethernet networking. Machines were grouped into two separate sub-networks that were connected through the Gigabit Ethernet networking.

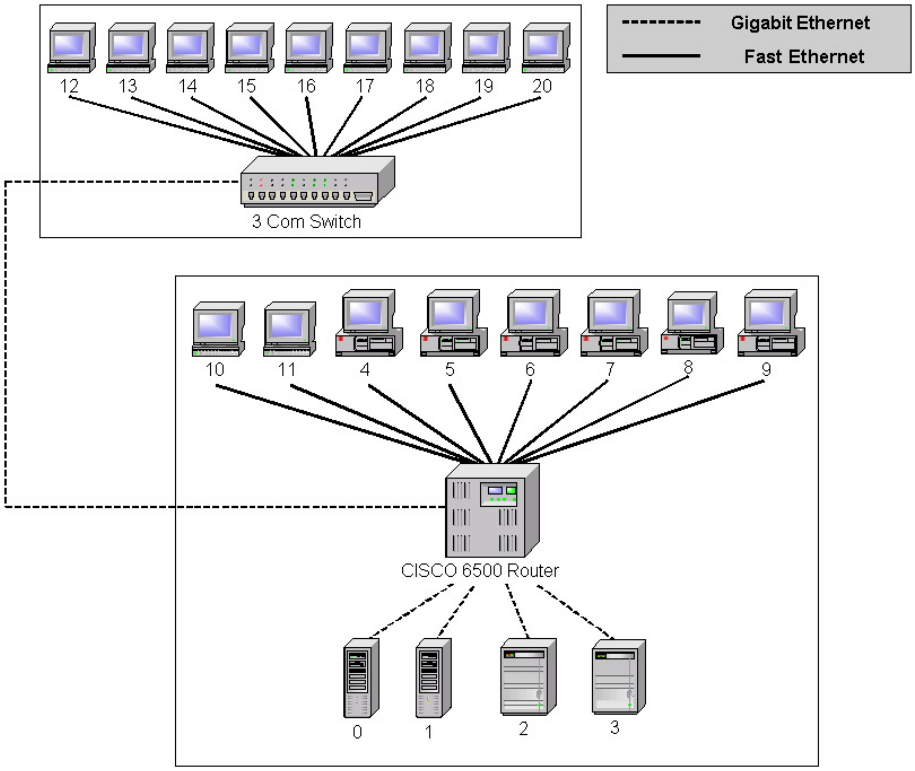


Figure 6.4: Heterogeneous Network Architecture

6.4 Heterogeneous Modeling

In chapter 4, we have developed an analytical model for *homogeneous* collection of processors. We extend that model to *heterogeneous* systems. Consider a collection of p processors, each with a different processing speed and memory. Recall that the computational time was defined as

$$T_{comp} = \frac{T_{par}}{P} r + T_{seq}$$

Let f_i be a distribution function that assigns the workload to each processor such that $\sum_p f_i = 1$. Workload assigned to each processor is determined by the load balancing techniques used. The relative speed of each processor, s_i, is given by the computation

time a processor can execute the workload. The computation time for each processor *i*, T_{comp}^{i}, can then be defined as

$$T_{comp}^{i} = \frac{f_i k T_b r}{s_i} + T_{seq}$$

This also correctly reduces to the homogeneous case, for which $f_i = \frac{1}{p}$ and the relative speed is 1 for all i.

Our experimentation testbed includes a set of heterogeneous networking technologies. The network topology, Figure 6.4, is arbitrary like Wide Are Network (WAN). In our experimentation all the fastest machines were on gigabit network and the slower machines were on 100BT network. Since the total execution time is primarily concerned with the slowest computer, we can use the network bandwidth of 100BT network in the communication model. We use the same communication model, which is

$$T_{comm} = T_{start} + T_W \frac{N}{P}$$

The total execution time then is defined as

$$T_{tot} = \frac{fkT_b r}{s} + T_{seq} + T_{start} + T_W \frac{N}{P}$$

6.5 Concurrent Sonar Processing

Concurrent sonar processing application and how computational resiliency is applied to it on homogeneous computing environment were described in chapter 4. Here, we extend the application to heterogeneous computing environment. We explore the performance

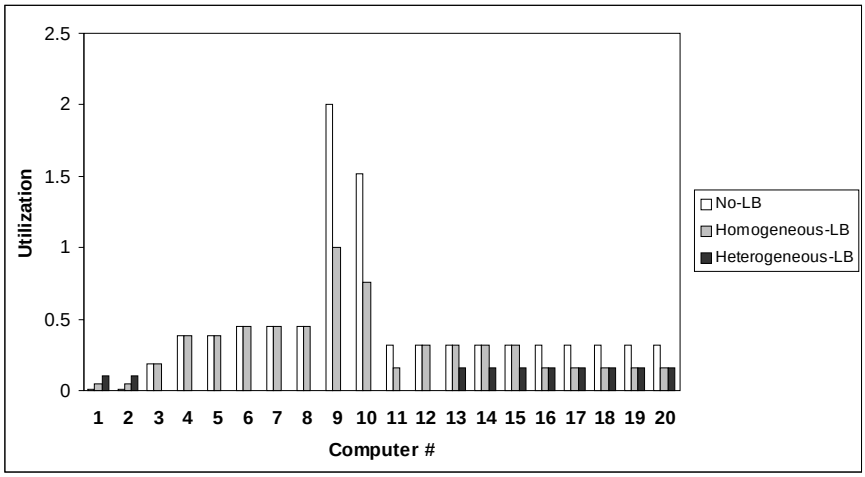
issues associated with various load balancing techniques and the overhead of resiliency. We show representative experimental results from a broad set of experiments that we have conducted to measure the effectiveness of load balancing and the overhead caused by resiliency and liveness checking on the heterogeneous testbed introduced in chapter 6.3.

The sensor and display were mapped to Machine 0 in the testbed due to memory concerns, while each of the remaining 20 machines executed beamformers. Resiliency was applied uniformly to harden the application by replicating the beamforming elements. The beamformer was executed once for Figures 6.5, 6.6 and Table 6.3, and 100 iterations for Figure 6.7. Each iteration processed a single set of buffered returns. Three parameters were varied in the experiments: the load balancing method, the level of replication (1, 3, or 7), and the frequency of the liveness checking (0 to 20 checks over the course of the 100 iterations). Even though resiliency 7 may seem to be a high level of replication, we consider this case interesting to investigate since it more closely approximates the computational model presented in chapter 1. The number of sonar elements and the number of buffered returns were fixed to 382 and 1000 respectively.

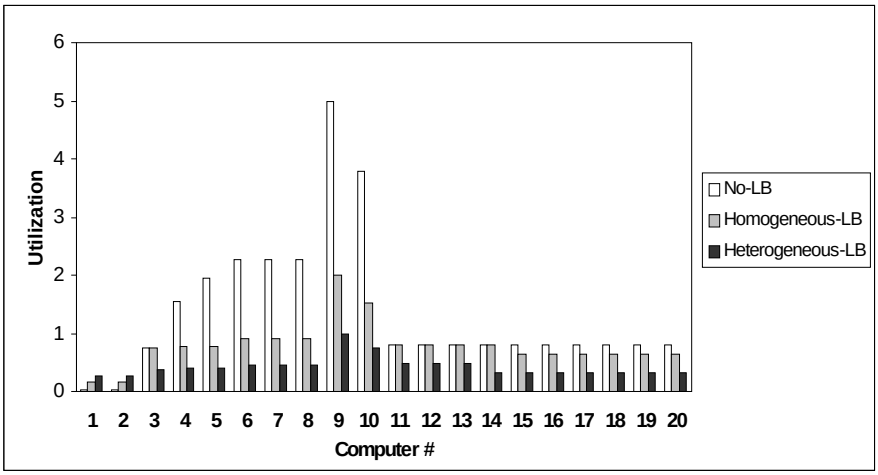
Three experiments were conducted to evaluate the effectiveness of the different load balancing techniques. First, the problem was run without load balancing (No-LB). Next, load balancing strategy based on the number of processors in each machine was used (Homogeneous-LB). In the third case, a small benchmark problem, roughly 20% as large as the full problem, was run on each machine to assess their relative performance. Using

static capacity estimates, the problem was then balanced (Heterogeneous-LB).

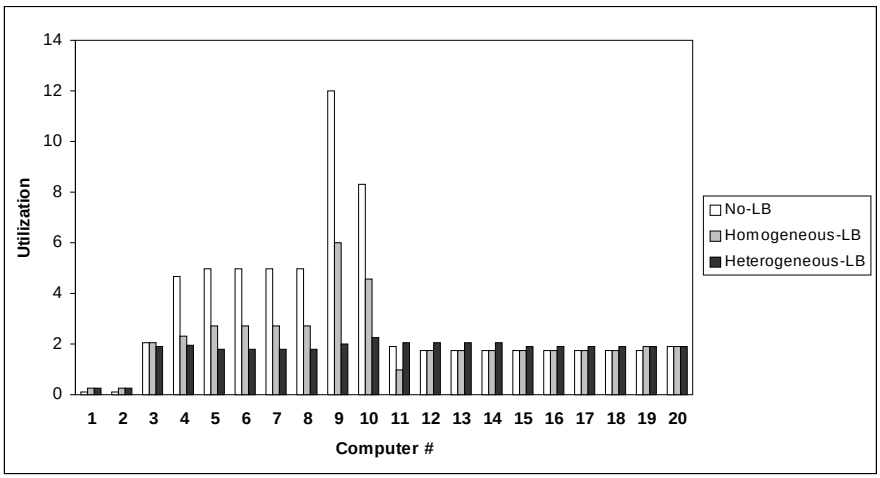
Figure 6.5 shows the relative utilization of the computers based on their relative capacity and workload assigned. For resiliency 1, it is permitted that no task is assigned to a slow processor, while in resiliency 3 at least one task is allocated to every computer to ensure higher reliability. For example, in Figure 6.5(a), machines 3 to 12 were dropped for resiliency 1. Heterogeneous-LB technique shows the most balanced utilization. Machines 1 and 2 have room to take more tasks but cannot due to the reliability constraints.



(a) Resiliency 1



(b) Resiliency 3



(c) Resiliency 7

Figure 6.5: Utilization for Each Load Balancing Technique

Table 6.3 summarizes the results of these experiments. With homogeneous-LB, a 1.9 fold performance improvement was observed with resiliency 7. With heterogeneous-LB, a 2.7 fold performance improvement was observed with resiliency 7.

Scenario	**Step Time (sec)**	**Improvement**
No LB	36.2	N/A
Homogeneous-LB	22.0	1.65x
Heterogeneosu-LB	16.1	2.25x

(a) Resiliency 1

Scenario	**Step Time (sec)**	**Improvement**
No LB	88.4	N/A
Homogeneous-LB	50.0	1.77x
Heterogeneous-LB	35.4	2.5x

(b) Resiliency 3

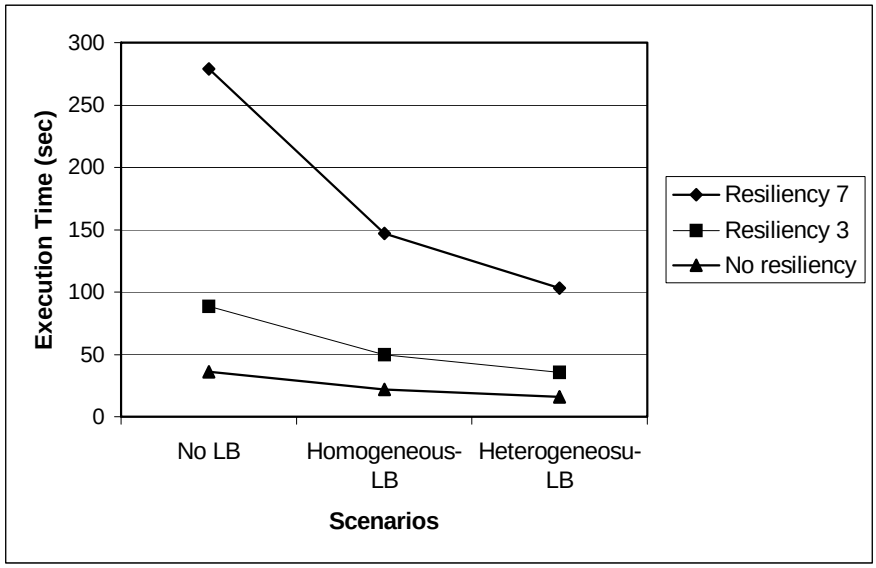
Scenario	**Step Time (sec)**	**Improvement**
No LB	278.9	N/A
Homogeneous-LB	146.9	1.9x
Heterogeneous-LB	103	2.71x

(c) Resiliency 7

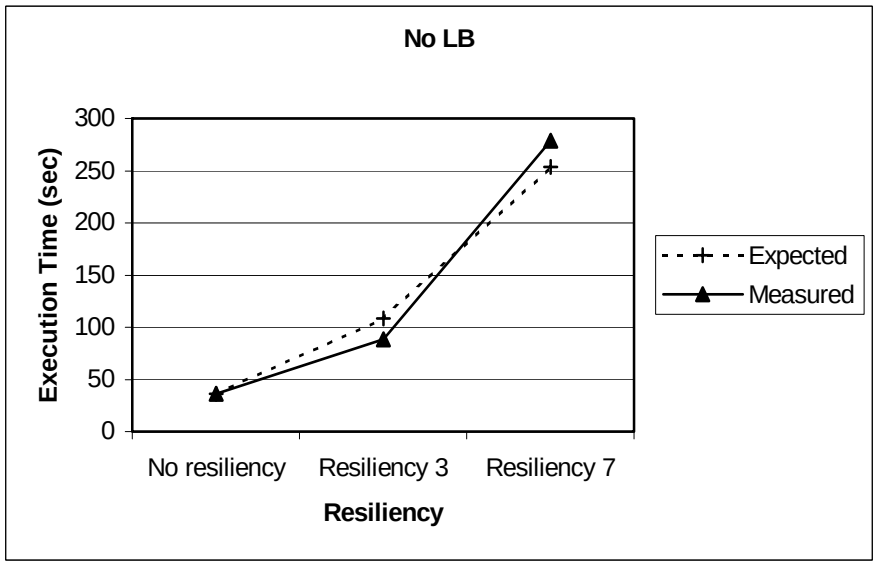
Table 6.3:Results of Load Balancing Experiments for Entire Heterogeneous Testbed

Figure 6.6 shows the overhead of resiliency with respect to each load balancing techniques. Our expectation was that since replication of a thread doubles its computational requirements, level 3 and 7 resiliency would execute with a three and seven-fold decrease in speed respectively. Without any load balancing, Figure 6.6(b), execution time for resiliency 7 increased more than a factor of 7. With load balancing, however, the results indicate that performance did not decrease linearly with the level of replication and was less than expected for all the cases. The execution time of resiliency 3

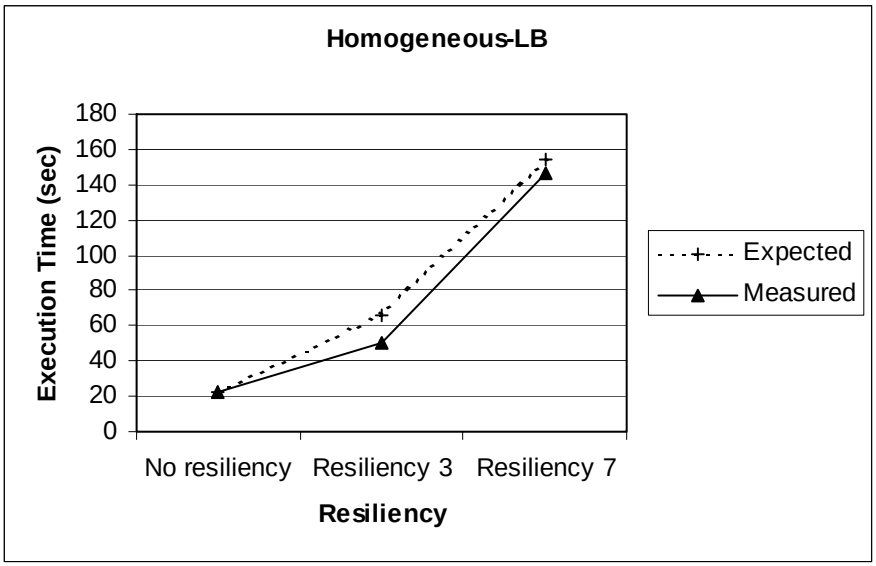
increased only 127% and 120% over resiliency 1 for Figure 6.6(c) and (d) respectively. For resiliency 7, it was as much as 568% indicating that very high levels of survivability may be possible without a direct linear cost. This artifact results from the overlapping of communication and computation in the resilient application: Idle time allowed cycles to be used in completing replicated tasks that would have otherwise been wasted. Obviously, this phenomenon is highly application dependent, however, idle cycles can occur for many reasons in distributed applications, e.g. file I/O, synchronization, global operations, etc. Therefore it is not unreasonable to assume that resiliency may often be achievable without significant computational costs.


300
250
200
150
100
50
0
Execution Time (sec)
No LB
Homogeneous-LB
Heterogeneosu-LB
Scenarios
Resiliency 7
Resiliency 3
No resiliency

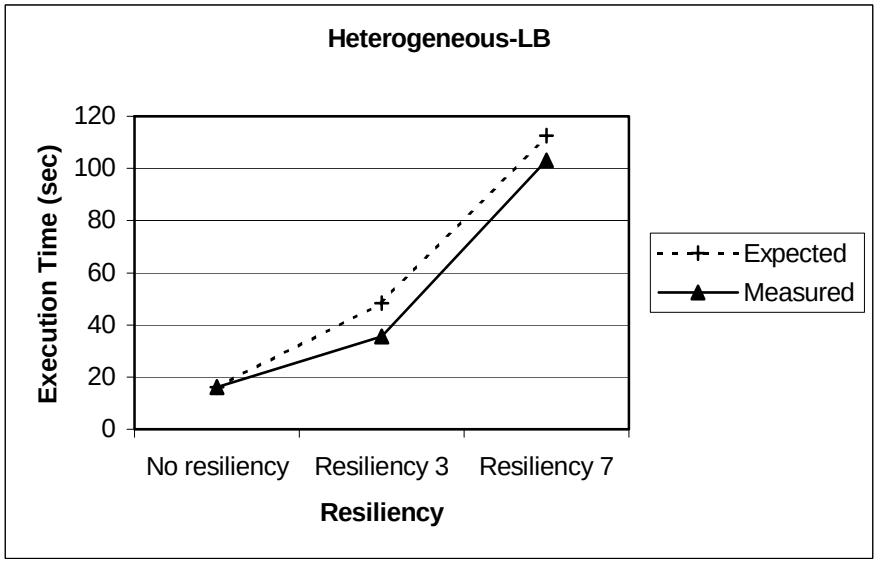
(a)


No LB
300
250
200
150
100
50
0
Execution Time (sec)
No resiliency
Resiliency 3
Resiliency 7
Resiliency
Expected
Measured

(b)



(c)



(d)

Figure 6.6: Overhead of Resiliency

Figure 6.7 shows the cost of liveness checking in this application. The overheads never exceeded 1% even when liveness checking is frequent (once every 5 iterations of the beamformer) and the level of resiliency is high, i.e. 7.

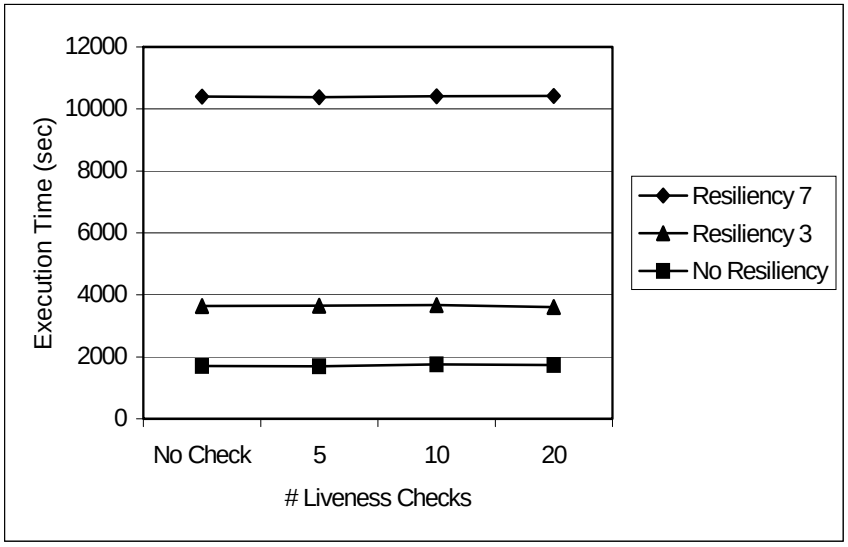


Figure 6.7: Overhead of Liveness Checking

While we have shown the effectiveness of the load balancing techniques for the large system with 21 computers in two sub-networks as above, it is also interesting to investigate the effectiveness of load balancing for a smaller system size. We repeated the same experiments on a small system with only 5 heterogeneous computers. Table 6.4 shows the processor information and Table 6.5 shows the results of load balancing. In this small sized system, we were only able to use lower degree of resiliency, 2 in this experiment. Experiment results show that the higher performance improvement was achieved. This small network case shows that higher performance improvement can be

achieved when the system is highly skewed, i.e., the difference between the fastest processor and the slowest processor is large.

	Processor Type	Memory Size	Operating System	Network (Mbps)	Relative Speed
0	450 MHz Pentium III (x4)	1.5 GB	Windows NT 4.0 Enterprise Server	1000	9.3 (x4)
1	500 MHz Pentium III (x8)	4 GB	Windows NT 4.0 Enterprise Server	1000	14.4 (x8)
2	500 MHz Pentium III	128 MB	Windows NT 4.0	100	2.6
3	200 MHz R4400	128 MB	IRIX 6.4	100	1
4	400 MHz Pentium II (x2)	256 MB	LINUX 2.2.12	100	3.1 (x2)

Table 6.4. Processor Information for small heterogeneous testbed

Scenario	Step Time (sec)	Improvement
No LB	180.8	N/A
Homogeneous-LB	83.5	2.17x
Heterogeneous-LB	49.0	3.69x

Table 6.5. Results of load balancing experiments for small heterogeneous testbed

6.6 Remote Sensing Application

The performance of another distributed application, remote sensing, was measured on the heterogeneous testbed environment. The same experiment was conducted with all workers replicated up to the level of seven; the manager and sensor were not replicated. Table 6.6 shows the results of load balancing experiments; these are consistent with those in the sonar application. Performance was improved by a factor of 3.37 for resiliency 7. As in concurrent sonar application, higher improvements were achieved with higher resiliency, i.e. 7.

Scenario	**Step Time (sec)**	**Improvement**
No LB	122	N/A
Homogeneous-LB	81	1.5x
Heterogeneous-LB	54	2.26x

(a) No Resiliency

Scenario	**Step Time (sec)**	**Improvement**
No LB	357	N/A
Homogeneous-LB	197	1.81x
Heterogeneous-LB	158	2.26x

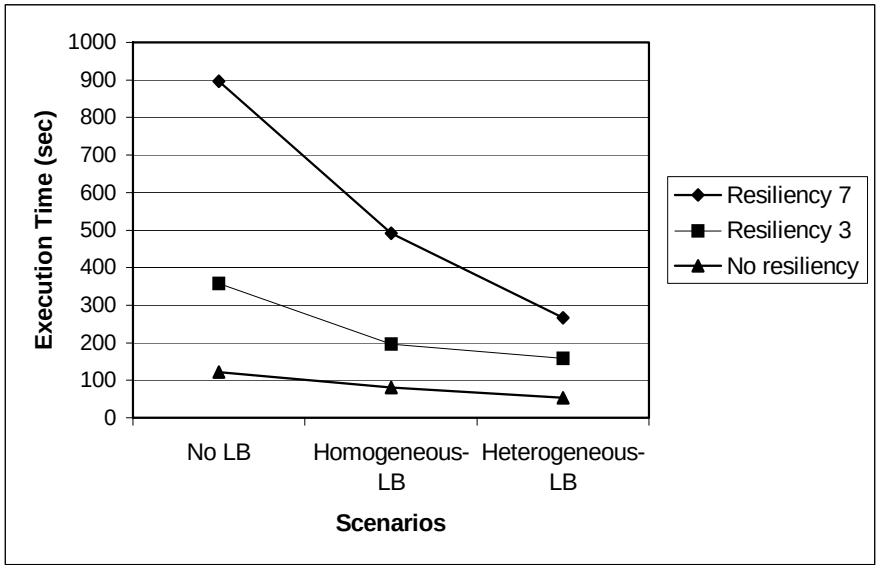
(b) Resiliency 3

Scenario	**Step Time (sec)**	**Improvement**
No LB	896	N/A
Homogeneous-LB	492	1.82x
Heterogeneous-LB	266	3.37 x

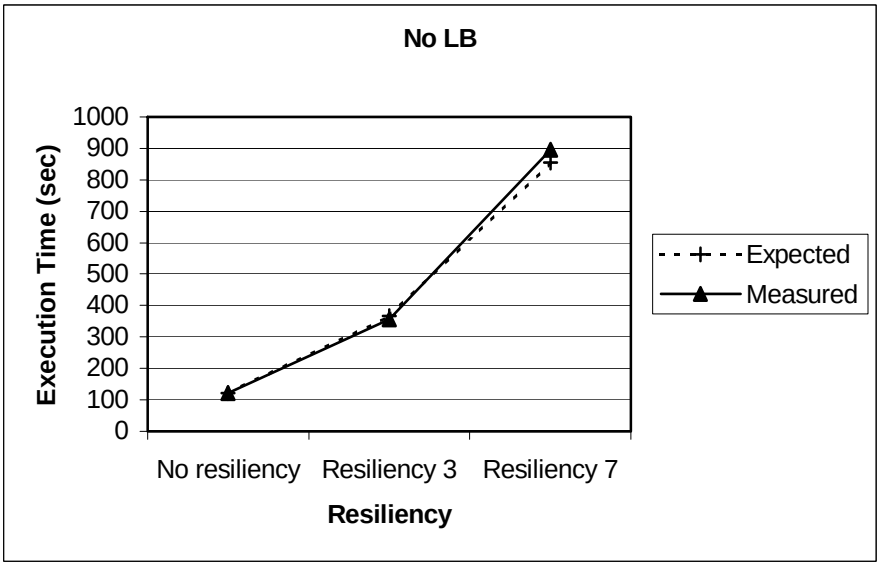
(c) Resiliency 7

Table 6.6:Results of Load Balancing Experiments for Entire Heterogeneous Testbed

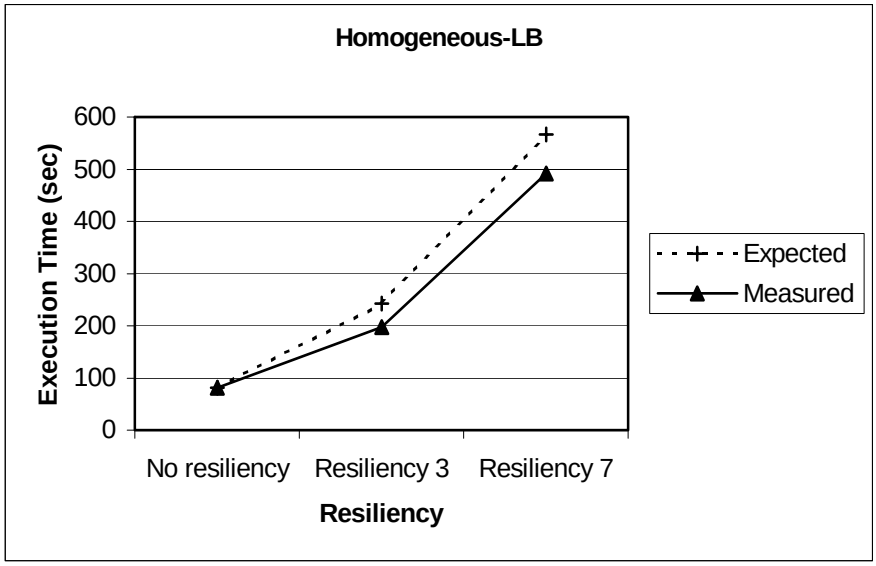
Once again, when resiliency was applied the expected result was that performance would decrease by a factor of three or seven depending on the specified resiliency since the replicated processes require both memory and processor resources. Figure 6.8 shows the overhead of resiliency with respect to three load balancing techniques. Without any load balancing, the execution times for resiliency 7 increased more than a factor of 7 in Figure 6.8(b). Heterogeneous load balancing reduced the overhead of resiliency significantly. With resiliency 7, the overhead was only 393% over resiliency 1 in Figure 6.8(d). As in the sonar application, we observe that load balancing improved the performance and resiliency is able to utilize idle cycles in the concurrent algorithm to reduce the cost of replication.


1000
900
800
700
600
500
400
300
200
100
0
Execution Time (sec)
No LB
Homogeneous-LB
Heterogeneous-LB
Scenarios
Resiliency 7
Resiliency 3
No resiliency

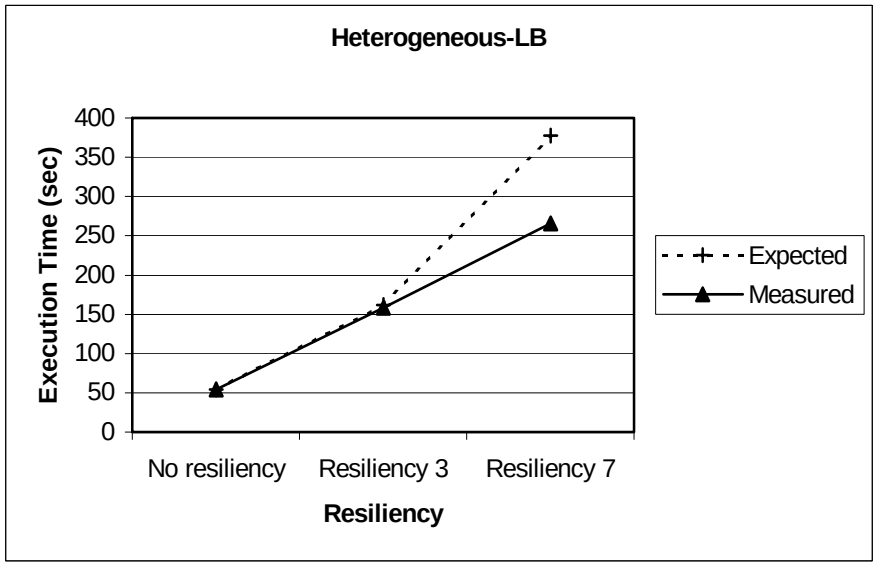
(a)


No LB
1000
900
800
700
600
500
400
300
200
100
0
Execution Time (sec)
No resiliency
Resiliency 3
Resiliency 7
Resiliency
Expected
Measured

(b)



(c)



(d)

Figure 6.8: Overhead of Resiliency

Figure 6.9 examines the overhead caused by liveness checking. In each case, the overhead was less than 1% and is consistent with the results from the sonar application.

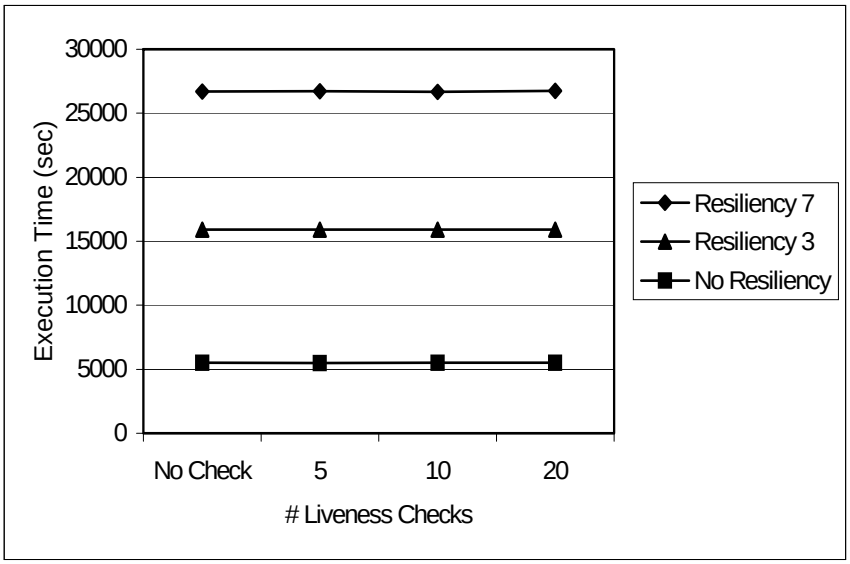


Figure 6.9: Overhead of Liveness Checking

The same experiment for small network size was conducted for resiliency 2. Figure 6.7 shows that use of load balancing produces performance improvement, which is better than large system case.

Scenario	**Step Time (sec)**	**Improvement**
No LB	1150	N/A
Homogeneous-LB	466	2.47x
Heterogeneous-LB	286	4.02x

Table 6.7. Results of load balancing experiments for small heterogeneous testbed

6.7 Summary

In this section, we have presented the load balancing algorithm to be used with computational resiliency. Experimental results show that the load balancing can improve the performance of the applications over the heterogeneous distributed environments. The load balancing algorithm was designed also towards improving the reliability of the system such that it meets reliability constraints. Experimental studies show that substantial improvements in performance are possible when one takes into account the individual resource capacities of the computers on which a concurrent application is running.

Chapter 7 Conclusion and Future Work

This thesis has described the notion of computational resiliency and presented the framework for developing highly reliable distributed computing systems. It discussed the implementation issues associated with a prototype-programming library that supports the idea. The thesis shows how the concepts and library can be applied in the context of two realistic military applications: a towed array sonar and a remote sensing application. The implementations of these applications were studied to ascertain the overheads associated with the technology on a homogeneous architecture. Then, they were further extended to heterogeneous architecture consisting of computers with varying computing capability, memory availability, operating systems, and networking technology.

For both applications, ability to utilize idle cycles to reduce the cost of increased survivability was evident, especially at higher levels of redundancy than one normally considers practical. This higher level is directly motivated by the computational model which provides strength in numbers. Although initially, the use of group based liveness checking was considered to be a significant defect with the current implementation strategy, it has proved to be less problematic than expected accounting for less than a 1% overhead in both applications. In both applications, reducing the frequency of checking could have reduced the overhead still further.

Use of load balancing techniques improved the performance by efficient allocation of the replicated threads on heterogeneous computing environments. Reliability was considered in the load balancing algorithm to improve the allocation of replicas. The results in this thesis indicate that the general concept is both practical and has less cost than originally anticipated

In order to understand the performance characteristics of the algorithm with computational resiliency, analytical models have been developed. Application dependent parameters and reliability requirements are described in the model for predicting practical properties such as runtime and resource requirements for two prototypical distributed applications. The outlined analytical models have been validated against a large set of experimental data on homogeneous architecture and heterogeneous modeling was developed.

Future work include developing camouflage and decoy technologies to provide higher survivability in more adverse environments in the respective of information warfare. In those technologies, the system assist processes in hiding themselves and in providing likely targets for attack. A simple form of camouflage involves changing the name of process in the process table, so that the application process is invisible to a casual inspection. More sophisticated camouflage will involve behavior: a process will have to take on more of the identity rather than just the name of the process it's impersonating. For example, a camouflaged process should consume roughly the same resources (memory, processor time) as the process being mimicked. Sometimes, it is not enough to

simply hide; it should provide a ready target so that an attacker will be fooled into thinking the attack has succeeded.

Appendix A Message Logging Based Approach

In this thesis, we presented a prototype implementation approach to computational resiliency that is characterized by no message logging, no use of network file system, and active replication. That provides a hot-start and roll-forward recovery scheme. One of the drawbacks of the current approach is global synchronization point during the liveness checking. We employed that technique to avoid roll-back in the presence of failure and saving intermediate states including messages. Another alternative approach is uncoordinated liveness checking with message logging in which the messages sent are kept until the destination threads receive them successfully. When failure happens, a new thread recreated in the destination thread group can request for the logged messages from the sender thread.

The major disadvantage of this approach is that it has to keep the copies of the messages until they become unnecessary. In general client/server application, where each transaction message size is not huge, this approach is applicable. However, for the application in which each message size is huge, for example 10Mbytes in remote sensing application, this approach suffers from severe I/O operations.

We have implemented this approach as an alternative implementation solution to computational resiliency and studied the performance of this approach. Experimentation results show that this approach incurs more overhead than our initial approach. The performance of the remote sensing application was measured on a distributed

environment consisting of 16 Sun Solaris 300MHz.workstations connected with 100BaseT networking technology. All workers were replicated to a level of two while the manager and the sensor were not replicated.

Figure A.1 shows the speed up gained as a function of the number of computers both with and without resiliency. Notice that the overhead caused by resiliency is approximately 10% plus the cost of replication uniformly. The concurrent algorithm operates within 20% of linear speedup in both cases. Theses preliminary results indicate that the message logging based approach causes more communication overhead than our initial approach where the resiliency overhead never exceeded the 100%.

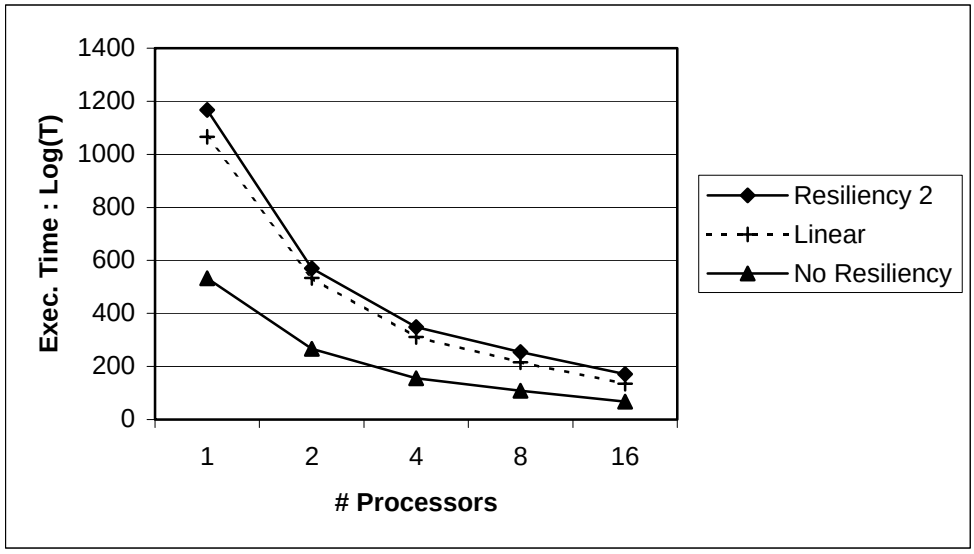


Figure A.1: Performance Chart

In our previous experimentations, the overhead due to resiliency never increased by the fold of resiliency. We believe the excessive overhead observed in this experimentation was due to message logging.

Appendix B Technology Demonstration

We have developed another technology demonstration program in fluid dynamics area, Dirichlet boundary problem. The Dirichlet boundary problem is a simple numerical simulation problem on a two dimensional grid. Each point on the grid has a (x, y) location and a value representing temperature of some material. At each time step, each point's temperature is averaged with its neighbor's temperatures to find the point's temperature at the end of the time step. The basic transition formula is:

$$Temp(x, y, t+1) = (Temp(x, y-1, t) + Temp(x+1, y, t) + Temp(x, y+1, t) + Temp(x-1, y, t) + Temp(x, y, t))/5$$

where $Temp(x, y, t)$ represents the temperature of location (x, y) at time t

This operates for all grid points that are not on the boundary. Boundary grid points are assumed to have a constant value. The Dirichlet problem is simple in that the workload is uniform. This allows the domain decomposition technique to be used in dividing up the workload among processes. A one-dimensional decomposition involves partitioning the grid by either rows or columns. A two-dimensional decomposition involves partitioning the grid by both rows and columns (i.e. into patches). In our explanation, two-dimensional decomposition is used. The Dirichlet boundary problem can be parallelized reliably using computational resiliency. Program B.1 describes the abstract code of what each node performs.

```
Entire two dimensional grid is decomposed into the patches
Computing node is created
Set up the communication channels among the nodes
For each node
begin
   Each node is assigned a patch with initial boundary condition
   Initialize each node state
   while not the norm is converged to an acceptable point
   begin
      for all neighbors
      begin
         send the edges to the neighbor
      end
      for all neighbors
      begin
         receive the edges from the neighbor
      end
      calculate the new temperature at each grid point of the patch
      calculate the new norm
      if (time expires) then perform liveness checking
   end
end
```

Program B.1: Dirichlet Boundary Problem and Its Parallelization

Each patch is assigned to a different processor for parallel computation. Figure B.1 shows how two dimensional grid is mapped into a matrix of processors, how each workload is assigned to a node, and how necessary communication paths are set up. In the right picture, each circle represents a node mapped to each processor and the arrows represent the communication paths among the processors.

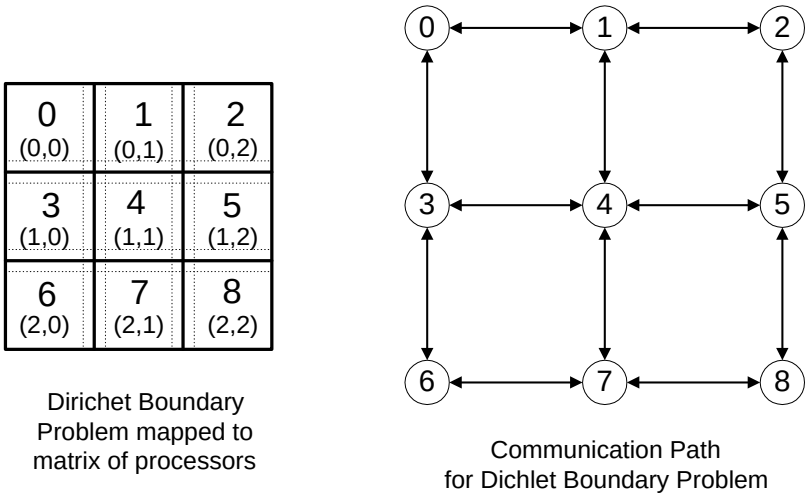


Figure B.1: Dirichlet Boundary Problem and Its Parallelization

Figure B.2 shows the actual snapshots of the screens. Right panel shows the computing status of the entire grid and the left panel illustates the current status of the system. As the failures happens, computer status window shows the crashed compyters and where the nodes moved. In this example, grid was divided into 2x2 pathes and four computers were used. Resiliency 2 was used uniformly.

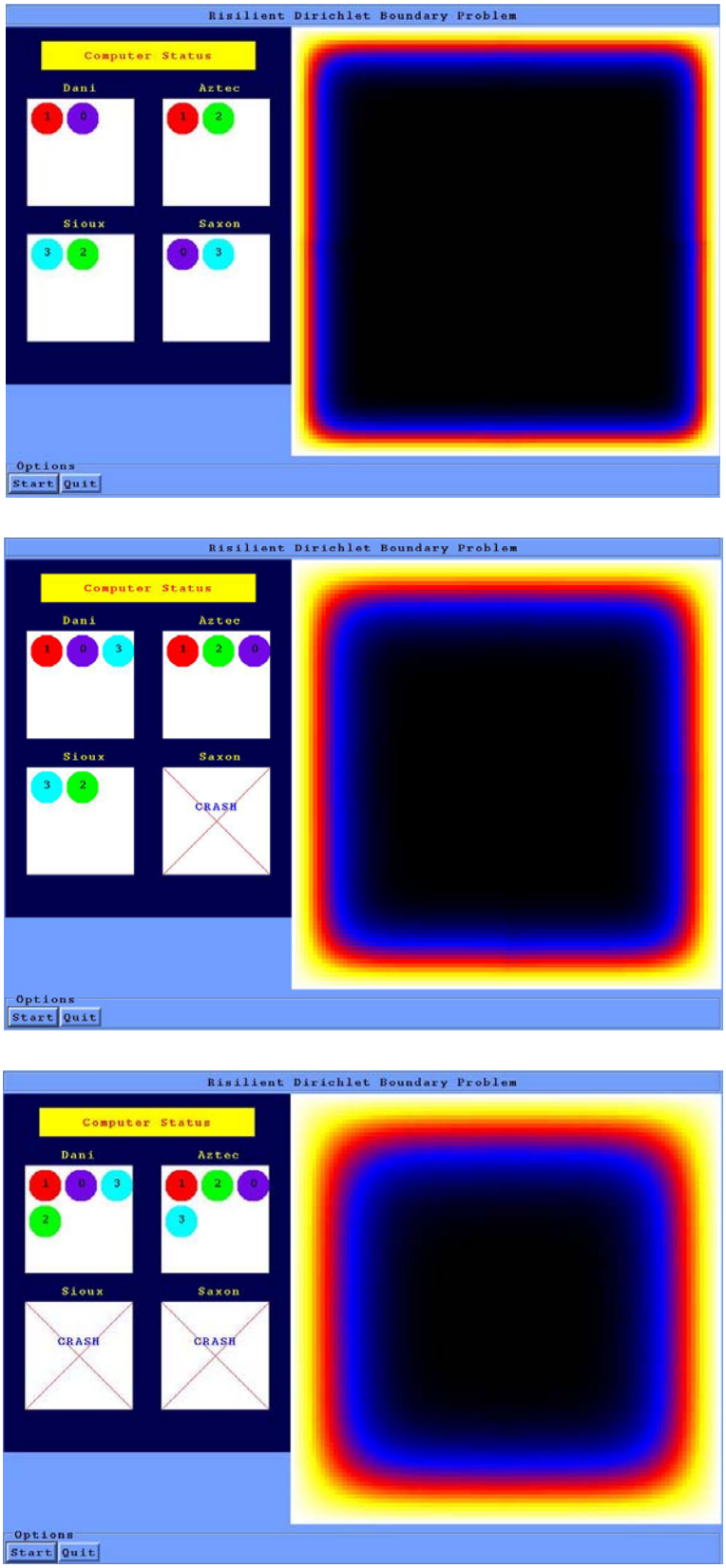


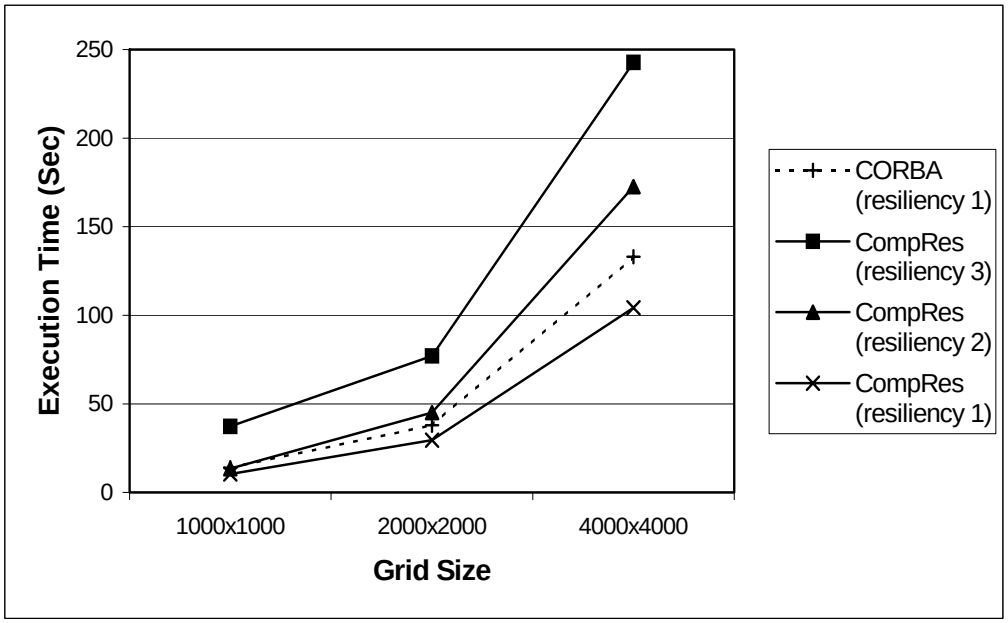
Figure B.2: Screenshots of Dirichlet Application Demonstration

Figure B.3 shows the experimental results with this application. It shows the the execution times of the application with varying problem size, the degree of resiliency, and the frequency of the liveness checking. It also compares the performance of computational resiliency with another frequently used distributed computing middleware, Common Object Request Broker Architecture(CORBA). CORBA is a middleware technology that provides a standard software specification for distributed object computing and seamless interoperability among heterogeneous clients and servers. It establishes the client-server relationships between objects such that a client can invoke a method on a server object regardless of the location of the object, programming language, and operating system. Even though CORBA has been used for many distributed computing applications, its performance is limited by communication overhead for marshaling/unmarshaling of the messages.

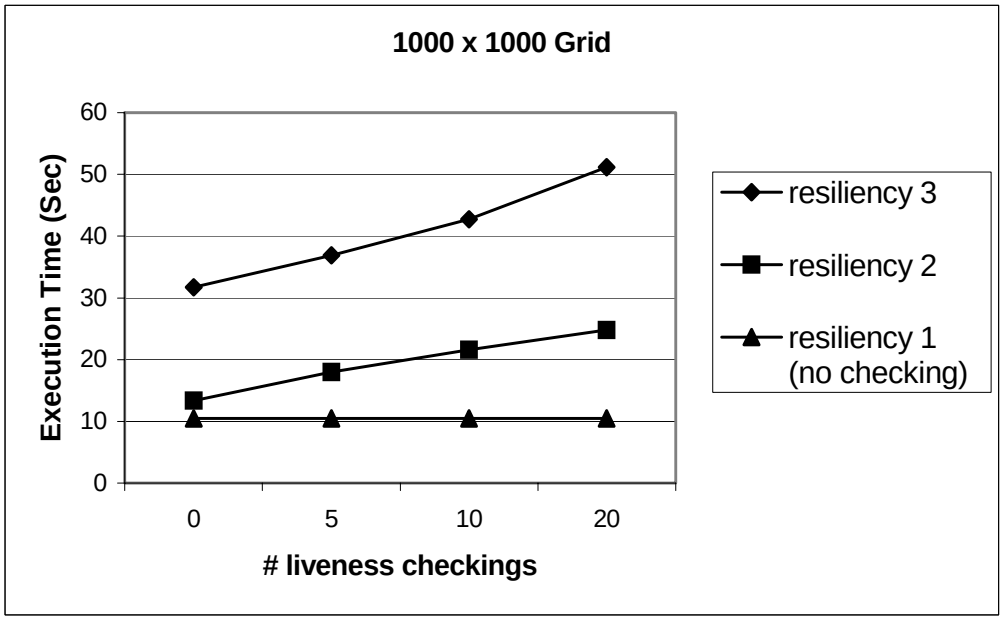
This experimentation was done over a cluster of four PCs running Windows NT with 128 Mbyte memory, 533MHz Intel Celeron processor, 128 Mbyte RAM, and 100BT network. The problem was decomposed into two by two patches and computation for each patch was mapped to each thread in each computer. One hundred iterations of computation were performed for each experimentation. In this experimentation, CORBA used JAVA native compiler for better performance.

In Figure B.3(a), CORBA had 30% overhead compared to computational resiliency with no replication in all three problem sizes respectively. Figure B.3(b), (c) and (d) show the execution times for varying problem size and resiliency levels. They indicates that the

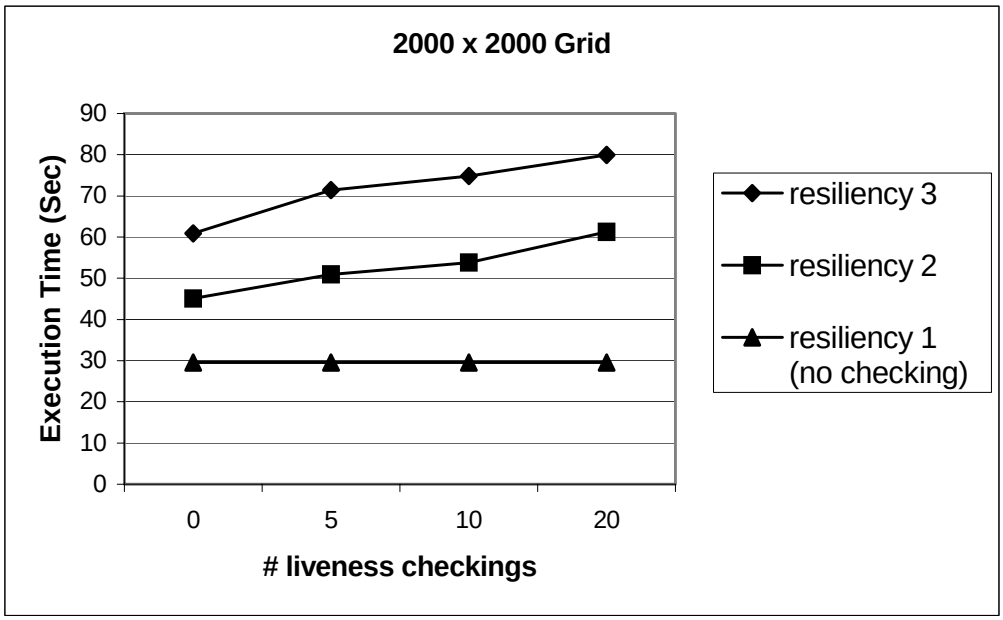
execution times didn’t increase linearly with the degree of resiliency applied as we saw in other applications in this thesis. The effect of the number of liveness checkings decreases as the problem size increases. For example, with resiliency 3 in 4000x4000 grid size the overheads for liveness checking for 5, 10, and 20 times were almost the same. For reasonably large problem size, the effect of liveness checking diminishes. For rather small problem sizes, liveness checking causes observable overhead as in Figure B.3(a).


250
200
150
100
50
0
Execution Time (Sec)
1000x1000
2000x2000
4000x4000
Grid Size
CORBA (resiliency 1)
CompRes (resiliency 3)
CompRes (resiliency 2)
CompRes (resiliency 1)

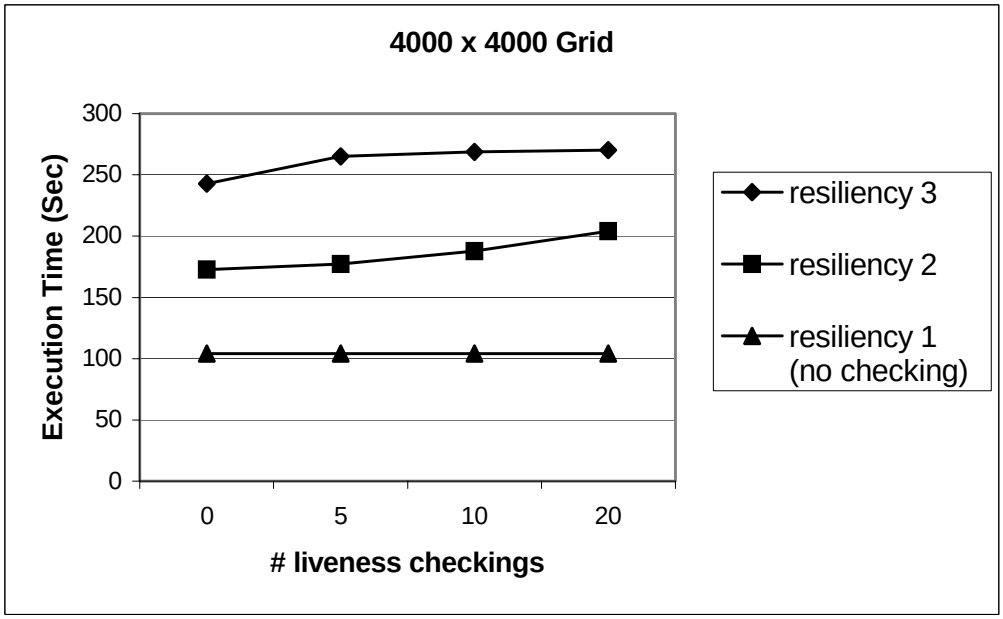
(a)


1000 x 1000 Grid
60
50
40
30
20
10
0
Execution Time (Sec)
0
5
10
20
liveness checkings
resiliency 3
resiliency 2
resiliency 1 (no checking)

(b)



(c)



(d)

Figure B.3: Performance Charts

Bibliography

1. Achalakul T., J. Lee, S. Taylor, “Resilient Image Fusion”, *Proceedings of the 2000 International Conference on Parallel Processing workshops*, Toronto, Canada, pp. 291-296, August 2000
2. D.A. Agarwal, “Totem : A reliable ordered delivery protocol for interconnected local-area networks”, *Ph.d dissertation*, Dept. of Electrical and Computer Engineering. University of California, Santa Barbara, 1994.
3. Amdahl, G. M., “Validity of the single-processor approach to achieving large scale computing capabilities”, AFIPS Conference Proceedings, vol. 30, pp. 483-485, 1967.
4. Anton H. and C. Rorres, *Elementary Linear Algebra: Applications Version*, John Wiley and sons, Inc., New York, NY, 1994.
5. Yair Amir, “Replication using group communication over a partitioned network”, *Ph.d Thesis, Hebrew University of Jarusalem*, 1995.
6. Y. Amir, Dolev., Kramer, S., and Malki, D., “Transis : A communication sub-system for high availability”, *Proc. of the 22nd Annual Internaltional Symposium on Fault-Tolerant Computing, pp 76-84*, July, 1992.
7. (a) Yossi Azar, “On-line Load Balancing”, *Proc. the 33rd Annual IEEE Symposium on Foundations of Computer Science, pp. 218-225*, 1992.
8. (b) Y. Azar, J. Naor, R. Rom, “The competitiveness of on-line assignments”, *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms, pp. 203-210*, 1992.
9. J.L. Baer, C. Girault, Cache coherence in MIMD systems: A Petri net model for a minimal state solution, *Van Nostrand Reinhold, New York*, pp292-328, 1990.

10. J.A. Bannister, K.S. Trivedi, “Task Allocation in Fault-Tolerant Distributed Systems”, *Acta Informatica, Vol. 20, pp. 261-281*, 1983.
11. S. Barnard, H. Simon, “A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems”, *Concurrency : Practice and Experience, vol 6*, pp. 101-117, 1994.
12. Thomas Barnard, *Radar and Sonar Signal Processing*, Unpublished, 1998.
13. K.P. Birman, van Renesse, R., “Reliable Distributed Computing with the ISIS Toolkit”, *IEEE Computer Society Press*, Lod Alamitos, Calif., 1994.
14. Bokhari, S.H. "On the Mapping Problem, “On the Mapping Problem”, *IEEE Transactions on Computers Vol C-30, No 3, pp 207-14*, March 1981.
15. Navin Budhiraja, Keith Marzullo, Fred B. Schneider, Sam Toueg, “Primary-Backup Approach”, *Proceedings of the Sixth International Workshop on Distributed Algorithms*, Haifa, Israel, 1992.
16. Chandy L. M., Taylor S., An Introduction to Parallel Programming, *Jones and Bartlett publishers*, Boston, 1992.
17. David R. Cheriton, Willy Zwaenpoel, “*Distributed Process Groups in the V-Kernel”*, ACM Transactions on Computer Systems, Vol 3, No 2, pp77-107, Feb. 1985.
18. Mark J. Clement, Michael J. Quinn, “Analytical Performance Prediction on Multicomputers”, *Proceedings of Supercomputing*, 1993.
19. D. Cohen, “On Holy Wars and a Plear for Peace”, *Computer, Vol. 14, No. 10, October, pp. 48-54,* 1981

20. T. E. Curtis and R. J. Ward, “Digital beam forming for sonar systems”, *IEE Proc.*, *Vol. 127, Pt. F, No. 4*, August, 1980.
21. G. Cybenko, “Dynamic Load Balancing for Distributed Memory Multiprocessors”, *J. Parallel and Distributed Computing, vol. 7, pp 279-301*, 1989.
22. G. Deconinck, “Survery of Checkpointing and Rollback Techniques”, *Technical Report 03.1.8 of ESPRIT Project 6731 (FTMPS),* May 1993.
23. D. Evand, W. Butt, “Dynamic Load Balancing Using Task-Transfer Probabilities”, *Parallel Computing, vol. 19, pp. 897-916,* 1993.
24. Thoman Fahringer, “Automatic Performance Prediction of Parallel Programming”, Kluwer Academic Publisher, 1996.
25. Ian Foster, *Designing and Building Parallel Programs*, Addison Wesley, 1994.
26. Geist A., A. Beguelin, and J. Dongarra, *PVM:Parallel Virtual Machine:A User’s Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994.
27. W. Gropp, E. Lusk, and A. Skjellum, “Using MPI: Portable Parallel Programming with the Message Passing Interface”, MPI Press, 1995.
28. Rachid Guerraoui, Andre Schiper, “Software-Based Replication for Fault Tolerance”, *IEEE Computer, pp68-74*, April 1997.
29. Alan Heirich, Stephen Taylor, “A Parabolic Load Balancing Method”, Caltech-CS-TR-94-13, Computer Science dept., California Institute of Technology, 1994.
30. Shushil Jajodia, Catherine D. McCollum, Paul Ammann, “Trusted Recvoery”, *Communications of ACM 42, 7*, July 1999.

31. David B. Johnson, “Distributed System Fault Tolerance Using Message Logging and Checkpointing”, Ph.D Thesis, Rice University, 1989.

32. Juang, T., S. Venkatesan, ‘Crash Recovery with Little overhead”, *Proceedings of the 11th international Conference on Distributed Computing Systems, pp454-461*, May 1991.

33. Kaashoek, M. F., A. S. Tanenbaum, K. Verstoep, “Group Communication in Amoeba and its Applications”, *Distributed Systems Engineering, vol. 1, no. 1, pp 48-58*, September 1993.

34. J. Kim, H. Lee, S. Lee, “Replicated Process Allocation for Load Distribution in Fault-Tolerant Multicomputers”, *IEEE Transactions on Computers, Vol. 46, No. 4, April, pp. 499-505*, 1997.

35. Koniges A. K., *Industrial Strength Parallel Computing*, Morgan Kaufmann Publishers, San Francisco, CA, 2000.

36. Richard Koo, Sam Toueg, “Checkpointing and Rollback-Recovery for Disitributed Systems”, *IEEE Transactions on Software Engineering, Vol 13, No 1, pp23-31,* January, 1987.

37. (a)Vipin. Kumar, A.Y. Grama, N. Rao Vempaty, “Scalable Load Balancing Techniques for Parallel Computers”, *J. of Parallel and Distr. Comp., vol. 22, pp. 60-79*, 1994.

38. (b)Vipin Kumar, Ananth Grama, Anshul Gupta, George Karypis, “Introduction to Parallel Computing”, The Benhamin/Cummings Publishing Company, Inc., 1994.

39. Lamport, L., Shostak, R., Pease, M., “The Byzantine generals problem”, *ACM TOPLAS Vol 4, No.3, pp. 382-401*, July, 1982.

40. (a) Joohan Lee, Stephen Taylor, “Advances in Computational Resiliency”, *IEEE Aerospace Conference, Big Sky, Montana, March,* 2001.

41. (b) Joohan Lee, Steve J. Chapin, Stephen Taylor, “Reliable Heterogeneous Applications”, *to appear in IEEE Transactions on Reliability*, Dec., 2001.

42. K. Li, J. Dorband, “A Task Scheduling Algorithm for Heterogeneous Processing”, *Proc. High Performance Computing, pp. 183-188,* 1997.

43. Jr. Wagner Meira, “Modeling Performance of Parallel Programs”, *Technical report 589, The University of Rochester, Computer Science Department, Rochester, New York 14627,* June 1995.

44. W. Keith Nicholson, *Elementary Linear Algebra with Applications*, PWS-KENT Publishing Company, 1986.

45. Netperf: A Network Performance Benchmark, http://www.netperf.org, Information Networks Division Hewlett-Packard Company, 1995.

46. Nielsen R., *Sonar Signal Processing*, Artech House, Inc., 1991.

47. N. J. Nilsson, *Problem Solving Methods in Artificial Intelligence*, New York: McGraw-Hill, 1971.

48. Noble B. and J. W. Daniel, *Applied Linear Algebra*, Prentice–Hall, Englewood Cliffs, NJ, 1988.

49. L.J.M. Nieuwenhuis, “Static Allocation of Process Replicas in Fault-Tolerant Computing Systems”, *Proc. FTCS-20, June, pp. 298-306,* 1990.

50. Pardalos P. M., A. T. Phillips, J. B. Rosen, *Topics in Parallel Computing in Mathematical Programming*, Science Press, New York, NY, 1992.

51. James S. Plank, Miach Beck, Gerry Kingsley, Kai Li, “Lipckpt : Transparent Checkpointing under Unix”, *USENIX Winter 1995 Technical Conference*, 1995.
52. Balkrishna Ramkumar, Volker Strumpen, “Portable Checkpointing for Heterogeneous Architectures”, *27th International Symposium on Fault-Tolerant Computing, pp 58-67,* Seattle, Washington, June, 1997.
53. Randell, B., “Reliable Computing Systems”, *Operating Systems:An Advanced Course*, Springer-Velag, New York, pp282-391, 1979
54. van Renesse, R., Birman, K. P., and Maffeis, S., “Horus: A flexible group communication system”, *Commumunications of ACM 39, 4*, Apr. 1996.
55. Ron Resnick, “A Modern taxonomy of high availability”, http://www.interlog.com/~resnick/ron.html, 1996.
56. Rieffel M., S. Taylor, J. Watts, and S. Shankar, “Concurrent Simulation of Plasma Reactors”, *Proceedings of High Performance Computing, Society of Computer Simulation*, pp. 163-168, 1997.
57. Rieffel M., S. Taylor, J. Watts, “Automatic Granularity Control for Load Balancing of Concurrent Particle Simulations”, *Proceedings of High Performance Computing, Society for Computer Simulation*, pp. 115-120, 1998.
58. Radu Rugina, Klaus Erik Schauser, “Predicting the Running Times of Parallel Programs by Simulation”, *International Parallel and Distributed Processing Symposium,* 1998.
59. D. J. Scales and M. S. Lam, “Transparent Fault Tolerance for Parallel Applications on Networks of Workstations”, *Proceedings of the 1996 USENIX Technical Conference*, January, 1996.

60. Fred B. Schneider, “Byzantine generals in action: Implementing fail-stop processors”, *ACM TOCS Vol 2,No. 2, pp 145-154*, May 1984.
61. Fred B. Schneider, “Implementing fault-tolerant services using the state machine approach : a tutorial”, *ACM Computing Surveys, Vol 22, No. 4, pp 299-319*, April 1990.
62. Seitz C.L., “The Cosmic Cube”, *Communications of ACM, vol 28, No. 1, pp 22-33*, 1985.
63. Serrano M. J., W. Yamamoto, R. C. Wood, M. Nemirovshy, “A Model for Performance Estimation in a Multistreamed Superscalar Processor”, *7th International Conference of Computer Performance Evaluation*, Vienna, Austria, May 1994.
64. S.M. Shatz, J.P. Wang, M. Goto, “Task Allocation for Maximizing Reliability of Distributed Systems”, *IEEE Transactions on Computers, Vol. 41, No. 9, pp. 1,156-1,168*, Sept., 1992.
65. Shivaratri, N. G., P. Krueger, and M. Singhal, “Load Distributing in Locally Distributed Systems”, *IEEE Computer, vol 25, no. 12, pp. 33-44*, Dec. 1992.
66. V. S. Sunderam, “PVM: A Framework for Parallel Distributed Computing”, Concurrency: Practice and Experience, Vol 2, No. 4, pp 315-339, Dec. 1990.
67. Jeremy B. Sussman, Keith Marzullo, *“Comparing Primary-Backup and State Machines for Crash Failures”*, Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, 1996.
68. Taylor S., Watts J., Rieffel M., and Palmer M., “The Concurrent Graph: Basic Technology for Irregular Problems”, *IEEE Parallel and Distributed Technology, 4(2): pp15-25*, 1996.

69. Taylor S. et al., Industrial Strength Parallel Computing, *Morgan Kaufmann, pp 147-168, 227-246, 267-296,* 2000.

70. (a) Watts J., M. Rieffel, S. Taylor, "Dynamic Management of Heterogeneous Resources", *High Performance Computing: Grand Challenges in Computer Simulation, April, pp.151-156,* 1998.

71. (b) Watts J., and Taylor S., "A Practical Approach to Dynamic Load Balancing", *IEEE Transactions on Parallel and Distributed Systems, vol 9, pp 235-248,* 1998.

72. (c) Watts J., Taylor S., and Nilpanich S., "SCPlib: A Concurrent Programming Library for Programming Heterogeneous Networks of Computers", *IEEE Information Technology Conference, EX 228, pp 153-6,* 1998.