FAULT TOLERANCE AND RELIABILITY ANALYSIS OF LARGE-SCALE MULTICOMPUTER SYSTEMS

by

Walid A. Najjar

Technical Report No. CENG 89-01

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

July 1988

To my parents

Acknowledgments

First and foremost, I would like to thank Professor Jean-Luc Gaudiot, my advisor, for the thorough guidance, constant support and encouragement he has provided throughout my years in graduate school. To him I express my deep feelings of respect and appreciation as a teacher, a researcher and a person.

I would also like to thank the members of my guidance and dissertation committees: Professors Kai Hwang, Dean Jacobs, Michel Dubois, Prasanna Kumar and Dr. David Mizell.

I would like to express my special gratitude to Dr. David Mizell for the constant support and valuable feedback he has provided.

My heartfelt thanks go to Dr. Fadi Kurdahi whose friendship and encouragements have been invaluable.

Finally, I would like to thank all the great friends and colleagues, both in the EE-Systems Department and at ISI, who have made these past years very enjoyable: Christoph Scheurich, Jean-Luc Jezzouin, Skevos Evripidou, Dr. Rivi Sherman, Steve Farnworth, Susan Coatney, Diane Demetras.

This work has been supported by the National Science Foundation under Grant No. CCR-8603772 and by the Office of Naval Research, Arlington, VA, under Contract No. N00014-86-K-0311.

Contents

List of Figures

List of Tables

Abstract

Recent developments in VLSI technology have made possible the design and implementation of massively parallel computing systems comprising several thousand processing elements. As the number of computing elements in a system increases, the likelihood of one or more elements failing, during a given time interval, as well as the complexity of the system level diagnosis increase. Fault-tolerance is, therefore, to become an integral part in the architectural design of large-scale systems and reliability an important measure in the evaluation of their performance.

This dissertation addresses the issue of the effects of increased processor failures rate in large-scale gracefully degradable distributed computing systems. A probabilistic model of network disconnection is developed and used to evaluate the effects of node failures on the network topology. The results show that although the probability of network disconnection decreases with increasing system size, the *resilience* of a given topology to network disconnection decreases when the connectivity is kept constant.

Combined measures of performance and reliability are used to evaluate the trade-off between increased computational power and failure rates as the number of processors is increased. It is demonstrated that, for a given recovery mechanism, there exists an optimal number of processors at which the amount of *reliable computational work* the system could deliver is maximum.

Finally, a simple distributed iterative algorithm for fault-tolerance is presented and evaluated. Based on a functional execution model of tasks, this algorithm allows the implementation of run-time fault detection, checkpointing and recovery.

Chapter 1

INTRODUCTION

The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible. It is not to be presumed that such an attempt has succeeded. How near the approach has been made must remain for aftertimes to determine.

Charles Babbage
On the Mathematical Powers of the Calculating Engine

The rapid development of Very Large Scale Integration (VLSI) technology, in the last decade or so, has opened a wide spectrum of new possibilities in the design of computing systems. In fact, in the past ten years, we have witnessed the development of ever more powerful and sophisticated commercially available microprocessors. While the switching speed of these devices has improved by several orders of magnitude, it has becomes more difficult and costly to achieve orders of magnitude speed-ups by solely upgrading the logic technology. More emphasis is therefore being put on building parallel architectures to achieve faster program execution.

The availability, at low cost, of fast microprocessors and high density memory chips has opened the way for the design and development of Large-Scale or Massively Parallel computing systems where thousands of Processing Elements cooperate on the solution of a single problem. Such systems are becoming a cost effective alternative to conventional supercomputing for a wide variety of applications. In fact, their cost-performance is roughly estimated at $1,000 per MFLOPS whereas it is on the order of $10,000 per MFLOPS in the supercomputer family.

Multiprocessors are generally classified as being *tightly coupled* or *loosely coupled.* Processors in a tightly coupled multiprocessor system communicate through a globally shared memory. In loosely coupled multiprocessors, or multicomputers, processors have their own local memory and communicate by message passing over a shared interconnection network. The distinction between message passing and shared memory systems is not a very strict one. In fact, several schemes have been proposed that allow the implementation of a *logically shared* global memory on a physically distributed local memory system [Ran87]. The Denelcor HEP [Kow85] and the BBN Butterfly [BBN85a, BBN85b] implement such a model.

The objectives in using multiprocessor systems can either be a higher throughput or faster execution time of a single application program. Higher throughput can be obtained with general purpose time-shared multiuser multiprocessors such as the Sequent Balance or Symphony systems, the Encore Multimax, the Alliant FX8, the BBN Butterfly, etc. Multiprocessors such as the Intel iPSC, the NCUBE, the Connection Machine, etc. are mostly intended for faster execution of application programs.

Large-scale computing systems are targeted towards applications that exhibit a very large degree of parallelism. The execution of these applications often requires a very large amount (thousands) of CPU hours which can become prohibitively expensive even on high-speed uniprocessor systems. Such applications are typically found in physics (e.g. quantum chromo-dynamics), chemistry (e.g. molecular modeling), aeronautical engineering (e.g. fluid dynamics), civil engineering (e.g. seismic modeling), signal and image processing, computer simulation, artificial neural networks, etc. By applying thousands of Processing Elements to a single job, the execution time can be reduced to a manageable size thereby allowing a more effective use of computer models. It is expected that the availability of massively parallel computing systems would computer models to be applied to very large problem sizes and therefore open new possibilities and yet unforeseen potentials in many areas of scientific research and engineering.

1.1 Large-Scale Computing Systems

The recent developments in VLSI technology has accelerated research in the area of massively parallel or large-scale computing systems (LSCS). The salient features of such systems, as described in [KR86], are:

- A large number of independent Processing Elements communicating over a high-bandwidth interconnection network.
- Highly distributed control and operating system functions in the processing nodes.
- Highly parallel applications that are modeled as a collection of concurrent and communicating tasks.

A commonly accepted criterion for defining massively parallel architecture is that of 1000 or more Processing Elements irrespective of the size or complexity of the individual processor. The objective behind such a criterion is to quantify the degree of parallelism available in the machine.

Two computing systems have been developed that can be categorized, because of their size, as massively parallel systems. The Massively Parallel Processor (MPP) was developed by Goodyear Aerospace Corporation for the Goddard Space Flight Center under a contract from NASA for satellite image processing applications. Started in 1979, the MPP was historically the first machine in this category [Bat80]. The MPP is an SIMD two dimensional array processor consisting of 16K (128 x 128) single-bit processing elements. The system can be configured as a flat array, a cylinder or a torus. More recently, the Connection Machine (CM) was built by Thinking Machines Corporation under contract from DARPA. Originally intended as a large-scale connectionist MIMD architecture for the simulation of entity-relationship graphs [Hil86], the Connection Machine, in its present form, is an SIMD architecture. It consists of 16K to 64K single-bit Processing Elements organized in groups of 16. Each group of 16 PEs are implemented on a single chip together with the associated section of the interconnection network. Groups of 16 PEs are connected in a binary n-cube. The Connection Machine is the first commercially available Large-Scale Computing System (LSCS). However, both the Connection Machine and the MPP systems have an SIMD architecture that relies on a central controller, the host computer, for instruction scheduling and I/O operations. Therefore, they do not exactly fit the above definition of large-scale computing systems.

The major research issues that face the development of massively parallel systems can be summarized in three main categories:

1. *Programmability:* the programming and execution models necessary to build software tools for a very large number of concurrent tasks.

2. *Communicability:* the necessary underlying communication structures that allows for fast transfer of information from one point in the system to another.

3. *Fault-Tolerance:* the ability of the system to sustain and gracefully recover from faults and failures.

While the development and architectural design of parallel computing systems is quickly becoming an established field, the research and development of the software tools necessary to program these architectures still lags behind. All the commercially available multiprocessor and multicomputer systems are programmed using traditional sequential languages (mostly C and FORTRAN and, to a lesser extent, Lisp) that have been augmented with a set of library routines. These routines implement the necessary operations such as process spawning, synchronization or message-passing. It is, therefore, the programmer's task to explicitly state and structure the execution of the program in parallel. Notable exceptions in this respect are *Lisp and C* [1] that were developed for the Connection Machine. Although *Lisp and C* are designed as extensions to existing languages, they allow implicit data parallelism.

Since the programmer cannot explicitly specify the parallel execution on several thousands processors, the programmability of LSCSs becomes a crucial issue in their development. The major task, in this respect, is to provide programming and execution models that allow implicit parallelism to be detected at compile time and exploited at run-time. The research, in this area, has focused on four programming and execution paradigms:

- *Functional Languages:* based on Lambda calculus, these languages offer a highly formal approach to programming. Examples include Lisp and FL [Bac78]. Reduction architectures have been proposed for the direct execution of such languages [Mag80].

- *Object-Oriented Languages:* these are are less formally defined than functional languages. As common characteristic, they offer the possibility to define data structures and functions at a higher level of abstraction than conventional languages. Examples include: SIMULA, the Actors model as proposed by Agha [Agh85], C++ [Str86], Linda [ACG86].

- *Data-Flow Languages:* they are closely related to functional languages. The main difference being in their data-driven execution model. Examples include: SISAL [MSA*85], Val [AD79], Id [NPA86].

- *Logic Programming Languages* exemplified by Prolog, their paradigm is based on predicate calculus.

The large number of Processing Elements makes it impossible to organize a large-scale system around a physically shared global memory. Therefore, any realistic LSCS should be organized around an interconnection network that allows communication between processors or clusters of processors. While low connectivity graphs such as the array and the torus are suitable for SIMD computations where all communications are made to near neighbors, their large diameter would introduce unacceptable delays in MIMD computations. On the other hand, low diameter graphs such as the binary n-cube have a very high node connectivity. Therefore, in large systems, each node

[1] *Lisp and C* are trademarks of Thinking Machines Corporation.

would have a very large number of links which can result in a high implementation cost per node. A family of interconnection network topologies suitable for large-scale MIMD computations called hypernets have been proposed by Hwang and Ghosh [HG87]. These are non-regular hierarchically structured graphs that exhibit a low diameter while preserving a lower node-connectivity.

As the number of Processing Elements is increased, the likelihood of one or more elements failing, during any given time interval, increases accordingly. The failure of an element has repercussions not only on the program execution, but also on the communication structure. Due to the expected increase in failure rate as the system size is increased, it is imperative that LSCSs be designed to sustain and gracefully survive failures. Fault-tolerance considerations, therefore, do not affect only the design of the individual Processing Element, but also that of the underlying communication network. Furthermore, the reliability analysis of these systems becomes an important element in their overall performance evaluation. The fault-tolerance issue in LSCSs, and the related reliability and performance analysis are thus the topic of this dissertation.

1.2 Issues of Fault-Tolerance

The advent of LSCSs places the issues of fault-tolerant design and reliability evaluation of multiprocessor systems under a new light. In fact, the presence of a very large number of processing elements within one multicomputer system not only increases the computational power of the system but also the likelihood of one or more processors failing within a given time interval.

As the number of processors increases, the execution time of a given computation, and therefore the total system utilization time, decreases. Because of the added parallelization overhead, the increase in speed-up is at best a linear function of the number of processors. On the other hand, the overall failure rate increases with the number of processors and with the complexity of the underlying interconnection network with respect to the failure rate of a single processor using a comparable technology. This increase is at least proportional to the number of processors and results in a decrease of the expected time to first failure of the system. Since both the computing power and the failure rate of the system are determined by the number of processors, and since this number, in LSCSs, is expected to be very high, the trade-off between performance and reliability is very critical to the design and implementation of LSCSs. In fact, one can readily see that, for example, in a system with 4000 processors where the mean time to first failure of a single processor is on the order of 10^5 hours, the expected time to first failure of the system is on the order of 25 hours. This value might often be smaller than the expected execution time of some computations.

Therefore, if LSCSs are to become a viable alternative to supercomputers for the execution of massively parallel computations, then fault-tolerance must become a primary objective in their architectural design and reliability a key issue in their evaluation.

The primary objectives of a fault-tolerant design in LSCSs are therefore to provide:

- correct computational results in the presence of failures;
- system level diagnosis;

The secondary objectives are:

- to maximize computational throughput
- to minimize the hardware and software overhead necessary to implement reliable execution

However, LSCS have an inherent redundancy that can be exploited in order to allow the system to gracefully degrade in presence of faults by providing continued operation at reduced performance levels. How to exploit this redundancy in order to provide an optimal compromise between reliability and performance is still an open problem. This problem is actually two fold:

- at the hardware level, the issue is the detection of the failure of an element and its subsequent isolation from the rest of the system, thereby allowing continued operation.
- at the software level, the issue is to restore a previously saved globally consistent state of the program and resume execution on a reduced system with, possibly, a balanced reallocation of the computing load across the system.

Because of the distributed nature of LSCSs, the fault-tolerance and reliability of the interconnection network are determining factors in the overall reliability of the system. In fact, the fault-tolerant capabilities are based upon the exchange of information between processing nodes and assume the ability of the underlying communication structure to carry out the required transfer of information across the system. Several interconnection structures have been proposed and researched [WF84]. Network topologies can be classified in two categories:

- *static* topologies, where the switching nodes correspond to the processing nodes such as in the hypercube topology;
- *dynamic* topologies, where the processing nodes are located at the input and output nodes of the network such as in a cross-bar switch;

The routing control in the network can be either:

- *circuit switched* where a physical path is established between source and destination nodes;
- *packet switched* where data is sent in one or several packets that includes the destination address;

In the present state of technology, it seems that static, packet switched, networks offer the most suitable approach for large-scale multiprocessing. This does not exclude that future technology developments might allow mixed mode networks that could incorporate, at different levels, static and dynamic topologies as well as integrated data and circuit switching.

The required feature of an interconnection network from the standpoint of system fault-tolerance and reliability can be identified as [KR86]:

- *Robustness:* is the inherent redundancy of communication paths provided by the interconnection network.
- *Reconfigurability:* a related notion which measures the ability of the communication structure to adapt to link or node failures by providing reliable communication between processors.

Robustness and reconfigurability are the primary requirements on a fault-tolerant and reliable communication structure. A related requirement is high diagnosability level. Diagnosis refers to the process of determining the location of a fault in the system. Correct diagnosis of faults is necessary to allow proper reconfiguration and recovery actions to be carried out. In a distributed system, the system level diagnosis is implemented by a testing structure where each node is tested by all or a subset of its neighbors. The test results are then shared by subsets of nodes to reach a coherent

conclusion on the state of the system. Kuhl and Reddy [KR81] demonstrated several algorithms for performing distributed diagnosis that can be proven correct for a maximum number of faults. They also proved that the diagnosability level achievable by a distributed fault-diagnosis algorithm is a direct function of the interconnection topology and the testing structure independently of any algorithm.

1.3 Dissertation Goal and Outline

The main goal of this dissertation is to study the issues of fault-tolerant designs and reliability analysis in large-scale multicomputer systems. The main characteristic that differentiates large-scale systems (with thousands of Processing Elements) form existing multicomputers or multiprocessors (with less than a hundred Processing Elements) is the system size. The impact of a very large system size on different measures of fault-tolerance will be the main focus of analysis throughout this dissertation.

The rest of this dissertation is organized as follows.

Chapter 2 presents a statement of the problem addressed in this dissertation as well as a review of the traditional approaches to fault-tolerant design and reliability evaluation. Proposed combined measures of performance and reliability are reviewed. The concept of distributed fault-tolerance is introduced. Finally, a summary of the major contributions made in this dissertation is presented.

Chapter 3 proposes a probabilistic approach to the evaluation of network fault-tolerance. A stochastic simulation of network disconnection is used to derive an approximate analytical model. The analytical model is then justified and verified for a family of regular graph topologies. *Network Resilience* is introduced as a new measure of the robustness attribute of interconnection networks and used to compare a number of regular and non-regular graph topologies.

Chapter 4 presents a computation oriented reliability evaluation of large-scale gracefully degradable systems. The measure of *Reliable Computational Work* is defined and used to demonstrate the non-scalability of graceful degradation.

Chapter 5 describes a methodology of functional program execution that allows asynchronous run-time checkpointing. An iterative distributed algorithm for fault-tolerance is proposed that allows for protection against failures as well as fault-detection. The performance and reliability improvements brought by this algorithm are evaluated and compared to other algorithms.

Concluding remarks and directions for future research in this area are presented in Chapter 6.

Chapter 2

PREVIOUS RELATED WORK

> *Two are better than one; because they have a good reward for their labor. For if they fall, the one will lift up his fellow: but woe to him that is alone when he falls; for he has not another to help him up.*
>
> Ecclesiastes, 4, 9-10

The design of fault-tolerant systems can be traced to the early years of computing systems design when John von Neumann, in 1956, described how redundancy can allow reliable systems to be built from unreliable components [vN56]. Since then, the evolution of fault-tolerant and reliable systems design has paralleled that of computer architecture through several generations of design technologies.

The research in this field has led to a better understanding of the types and characteristics or faults and failures that can affect the operation of a computing system [SS82]. Three major categories can be outlined according to the origin of the fault or failure, as:

1. *Design errors:* these are either hardware or software design errors that are more manifest in the prototyping and initial stages of a product. However, some of these might not be detected before years of utilization.

2. *Component failures:* an electronic component might fail because of overheating, current overdriving, aging, etc. They generally result in a permanent failure of the system.

3. *Transient faults:* these are intermittent faults of random nature whose effect, as opposed to the previous two types, are not reproducible. They originate from non-controllable sources such as cosmic radiations, electromagnetic noise (particularly in pulse form), etc.

Studies have shown that non-permanent faults account for 90 to 98 % of all detected faults [MST79]. Furthermore, Fuchs *et al.* argue in [FAH83] that the increase in integration level, results in reduced voltage levels and a subsequent reduction of noise margins which might increase the susceptibility of VLSI circuits to electromagnetic noise interference.

The purpose of a fault-tolerant design is therefore twofold:

1. To guarantee continued system operation in the presence of failures. This implies the detection of, and subsequent recovery from, permanent failures.

2. To provide a high probability of correct computational results and therefore the protection against non-permanent faults.

Redundancy, in various forms, has been the basis of most, if not all, fault-tolerant designs techniques and methodologies. In general, redundancy techniques call for the multiple execution of a computation; the majority result is then taken to be the correct result. Redundancy techniques can be applied either in *time* or *space*. In time redundancy, the same computation is executed repetitively until a majority result is obtained [PF82]. While this technique eliminates the effects of intermittent or transient faults, permanent faults cannot be detected unless different hardware units are used for the different executions. Space redundancy, commonly known as N-modular redundancy (NMR scheme), calls for N copies of a computation to be executed, concurrently, on N distinct hardware elements. The majority result is determined using a voting scheme. In order to guarantee a strict majority, N is usually chosen as an odd integer. Several variants of the N-modular redundancy scheme, such as reconfigurable NMR, hybrid redundancy and backup sparing, are described in [BF76, SS82].

In this chapter we review the main concepts and ideas in fault-tolerant design and reliability evaluation that relate directly to the design and evaluation of LSCSs.

2.1 Traditional Approaches to Fault-Tolerance

With the evolution of computing systems, fault-tolerant design has developed along two distinct directions satisfying the following objectives:

1. *Mission-Oriented Applications* where the objective is to guarantee a system reliability above a given minimum level for the duration of a mission. Such systems are often called *ultra-reliable* and are intended for situations where repair is not possible such as in space and military applications.

2. *Availability-Oriented Applications* where the objective is to maximize the percentage of time the system is up. These systems, therefore, aim at minimizing the frequency and duration of necessary repairs. The main applications of such systems are in On-Line Transaction Processing (OLPT) such as used in banking transactions and airline reservations. Historically, the earliest systems designed for high availability are computer-controlled electronic switching systems used in telephone networks.

Examples in the first category are the STAR [A*71] developed at the Jet Propulsion Laboratory for NASA. The STAR is a self-testing computer system that implements repair by switching in redundant spare units. The FTMP [HSL78] and the SIFT [WLG*78] were two concurrent projects aimed at designing an ultrareliable airline guidance system with a failure rate lower than 10^{-9}. The FTMP employs a form of redundancy related to the TMR-Hybrid redundancy called Parallel-Hybrid redundancy, in which each major module can substitute for any other module of the same type. The SIFT is also based on the replication of basic components but relies upon the software to detect and analyze errors and to dynamically reconfigure the system to bypass faulty units.

Several high-availability systems that fall in the second category are now commercially available [Ser84]. Examples include: the Nonstop system from Tandem Computers, the Synapse N+1 from Synapse Computer, etc. The ESS, from AT&T, is intended for the control of electronic switching systems [Tro78]. Based on a dual-processor architecture, the ESS is designed for a requirement of less that two hours of downtime in a 40 years period.

In both of these approaches redundancy is extensively used to improve the reliability, and/or availability, of the system. In LSCSs, the available hardware redundancy is aimed primarily at

increasing the performance of the system. However, because of the potential for an increased rate of failures in the system, this redundancy can be used to provide continued system operation in the presence of failures, albeit, at the cost of a decrease in system performance, thereby allowing the system to gracefully degrade.

2.2 Network Fault-Tolerance

Because of the highly distributed nature of LSCSs, the reliability of the communication structure acquires an increased importance. In fact, the loss of communication between any two nodes might halt the execution as well as impair the system's ability to rollback and recover. The design and performance analysis of interconnection networks for multicomputer systems has been a very popular area of research. Extensive research has been done in the design of high performance connection networks for very large number of processing nodes [Fen81]. The emphasis, in these designs, is on high performance, reliability and diagnosability of the systems as allowed by the interconnection network.

As outlined in section 1.2, the basic requirements on an interconnection network, from the system fault-tolerance standpoint are: (1) robustness and (2) reconfigurability. Robustness is a notion tightly related to the inherent redundancy available in the network topology itself. For example, a simple ring topology would allow up to two, non-adjacent, node or link failures after which the system is partitioned and routing across the system is impaired. Reconfigurability, on the other hand, relates to the ability of the routing algorithm to exploit the inherent path redundancy and provide reliable communication between processor pairs in the presence of node or link failures.

Several protection schemes specific to some network topologies have been proposed and analyzed. Raghavendra *et al.* [RAE84] propose a reconfiguration scheme for binary-trees based on spare Processing Elements, the number of spares being $\log N$ where N is the number of active Processing Elements. Rennels [Ren86] proposes a similar dynamic strategy for binary n-cube topologies (hypercubes). Fortes and Raghavendra [FR85] describe a dynamic reconfiguration scheme, with no spares, for array architectures, based on alternating row and column elimination. Pradhan, [Pra85b, Pra85a], describes a class of reconfigurable and optimally fault-tolerant network architectures. Similar networks have been described in [EH85] and [SSB87].

When the application is not topology specific, the loss of a Processing Element can modify the network configuration. Since this configuration is not, in the general case, a fully connected graph, successive failures can eventually result in a disconnection of the system, and therefore prevent some processors from communicating to some other processors. In systems that rely on a distributed fault detection, recovery and restart procedure, the connectivity of the graph is crucial to the success of this procedure. Pradhan has proposed the measure of *network fault-tolerance* as the number of processors that can fail while preserving graph connectivity [Pra85b]. Therefore, in a regular graph with connectivity n, the network fault-tolerance is $(n-1)$. In the general case of a non-regular graph where the connectivity of node i is denoted by n_i, network fault-tolerance can be defined as:

$$NFT = \min_i(n_i)$$

Network fault-tolerance is a very conservative measure of robustness that does not take into account the size of the system and therefore the actual *probability* of occurrence of a network partition as a result of node or link failures.

2.3 Evaluation Measures

The traditional fault-tolerance measures of reliability, availability and mission-time do not provide an adequate evaluation of degradable systems. These have been devised for ultrareliable (mission-oriented) or highly available systems and therefore fail to take into account the capability of multiprocessors to provide continued operations at lowered performance levels.

New measures, therefore, have been devised that provide a combined performance/reliability evaluation of degradable fault-tolerant computing systems. A conceptual framework of composite performance and reliability measures was first proposed by Meyer in [Mey80]. This framework is general enough to accommodate all possible forms of *performance accomplishments* or *rewards* such as throughput or computational work.

The performability of a system S is formally defined with respect to a set of accomplishment levels A as follows [Mey80]:

Definition 2.1 *If Y_S is a performance measure of a system S taking values in the set A, then the performability of S with respect to a subset B of A ($B \subseteq A$) is defined by the function $p_S(B)$ as:*

$$p_S(B) = Prob(\{\omega \mid Y_S(\omega) \in B\})$$

In other words, given a performance measure (such as throughput) denoted by Y_S with a range of possible values A. The performability of the system S is the probability that the performance measure of the system after an event ω will be in a subset of A denoted by B. As an example, the performability of a degraded system could be the probability that its throughput be larger than 75% that of the undegraded system.

Closed-form solutions methods of performability for different types of systems have been proposed in [Mey82, FM84, DB87]. More recently, Smith *et. al.* have proposed a general framework of performability models based on Markov Reward Models (MRM's) [STR88].

Beaudry proposed the measures of *Computational Availability*, which is the expected value of the computation capacity of the system and *Capacity Threshold* which is the time when a specific value of the computational availability is reached [Bea78]. Performability and computational availability are measures that evaluate the computational capacity of a system in time and therefore are ideal for throughput oriented applications such as on-line transaction processing.

Also proposed by Beaudry is the *Computational Reliability* measure, which is the probability that at a given time the system correctly executes a given task, and the *Mean Computation Before Failure* ($MCBF$) which is the expected amount of computational work the system can deliver before the first failure, [Bea78]. These two measures are more suitable for *computation-oriented* applications where the critical issue is the *reliable completion* of the computation as is the case with massively parallel applications.

2.4 Distributed Fault-Tolerance

Since the very large number of PEs in a LSCS precludes the reliance on any physically shared global resource such as memory or controller, it implies that no central device can be used for the testing and monitoring of the system. Therefore, due to the distributed nature of LSCS, the task of identifying and isolating faulty PEs must itself be distributed throughout the system. In fact, a central controller or monitor would not only be a single point of failure in the system but might also become a performance bottleneck due to the large number of PEs. Preparata *et*

al. [PMC67] proposed a fault-tolerance algorithm where diagnosis methods are used to detect faulty elements that are subsequently isolated from the system. Based on this scheme, Kuhl and Reddy introduced the notion of *distributed fault-tolerance* as an attempt to allow the diagnosis and subsequent isolation of failed processors without relying on any central controller or memory system [KR80]. A distributed fault-tolerance allows for the graceful degradation of a system by providing for:

1. the detection of failures through system level diagnosis,
2. the isolation of the failed element from the rest of the system through logical and/or physical system reconfiguration,
3. the recovery from the failure and the restart of the computation. This step assumes the existence of some form of checkpointing that allows the rollback to a safe state in the computation.

For a survey of fault-tolerance issues in large-scale systems, the reader is referred to [KR86].

Several fault-tolerant schemes have been proposed that provide correct operations and allow the detection of faulty elements. In the general case of unstructured systems, the problem of *distributed* system diagnosis is known to be NP-complete. The system is therefore partitioned into a set of more manageable subsystems [Mal80, MM82]. A system is said to be t-diagnosable when up to t faulty elements can be detected, a comparison of modularly redundant and t-diagnosable systems is given in [CH81]. The proposed scheme is based on the assumption that each task is equivalent to a complete test (hundred percent fault coverage) of a processor. A recursive fault-tolerant algorithm, based on combined space and time redundancy, is proposed in [Agr85]. In this algorithm a task is initially executed on two processors and their results are compared, in the case of a mismatch, the task is executed on a third processor and the results are compared to the previous two, this process is recursively repeated until a match occurrs. The redundancy in this case is used to obtain a *plurality* vote. This algorithm allows for fault detection and correct task execution while maintaining a high system throughput. This scheme, however, relies on central commonly shared units such as a scheduler and signature processor as well as on recursively built common data structure that holds all the previous results pertaining to each task.

The Token Resending scheme, proposed by Gaudiot *et al.* [GR85], is based on the full replication of data and task restarting in a data-driven execution model. This scheme exploits the functional properties of data-flow execution to insure the retriability of tasks. Input data tokens to any actor are preserved in duplicates until the acknowledgement of the successful completion of that actor is received. This scheme, therefore, provides a the possibility of run-time checkpointing. Its implementation on a Tagged-Token data-flow architecture is described in [NG87a]. These features will be discussed in more details and exploited to implement distributed fault-tolerance in Chapter 5.

A similar scheme for distributed recovery has been proposed by Lin and Keller [LK86] for applicative program execution. It assumes a tree-structured execution of program tasks where no cycles are allowed.

2.5 Contributions of this Dissertation

In addressing the general issue of fault-tolerance and reliability analysis in large-scale multicomputer systems, this dissertation makes the following contributions:

1. *Disconnection probability.*
   An analytical model is devised that evaluates the probability of a network partition occuring as a result of multiple node failures. This model is verified by simulation results.

2. *Measures of network fault-tolerance.*
   Two new measures of network fault-tolerance are introduced: *Network Resilience* and *Relative Network Resilience.* These measures allow a comparison of different network topologies based on the cumulative probability of network disconnection.

3. *Effects of system size.*
   An analysis of the effects of a very large system size on reliability measures is presented. We demonstrate, analytically, the non-scalability of graceful degradation in large systems.

4. *Computational mesures of reliability.*
   The measure of computational work is used to evaluate the performance of a system as function of its size. The results show that an increase in system size might result in a decrease in the probability of safe completion of a lengthy computation.

5. *Distributed chekpointing.*
   A scheme, based on functional task execution, is proposed that allows the run-time checkpointing of programs with minimal overhead.

6. *Distributed fault-tolerance algorithm.*
   Based on the checkpointing scheme, an algorithm is proposed that implements distributed fault-tolerance as well as system level diagnosis. Its performance evaluation shows it to be superior to similar algorithms under normal conditions.

Chapter 3

A PROBABILISTIC EVALUATION OF NETWORK FAULT-TOLERANCE

"Look, Dave ... If you check my record, you'll find it completely free from error."

"I know all about your service record, Hal-but that doesn't prove you're right this time. Anyone can make mistakes."

"I don't want to insist on it, Dave, but I am incapable of making an error."

Arthur C. Clarke
2001: A Space Odyssey

The interconnection network is a central and major feature of any large-scale system design. Its characteristics determine the speed and volume of possible data communication which, in turn, determines its expected performance. They also determine the size of a realizable system in a given technology. As noted in Chapter 1, a large system size increases the probability of failures. Multiple node or links failures in a communication structure that is not based on a fully connected graph (as in a single bus based system) might result in a partitioning of the system. This event not only prevents the transfer of information between any two pairs of processors, but would also impair the process of distributed recovery and therefore result in a total system failure. In this chapter we propose a probabilistic approach to the analysis of network disconnections in distributed systems. The approach is based on a mixed analytical and simulation evaluation of the disconnection probability in a given network topology.

The assumed system and processor model is shown in Figure 3.1. It consists of a set of processing elements (PEs) communicating over an interconnection network. Each PE has a finite set of individual links to the network. The major components of a PE are a local memory system, a Processing Unit (PU) and a Communication Unit (CU). All communications from the PE to the outside world are handled by the CU over the set of links. In the analysis that follows, we will assume that the failure of either the PU or the CU is equivalent to the failure of all the links and therefore results in a complete PE failure. Furthermore, we will assume that the failures are uniformly and independently distributed throughout the system. This means that all processors are equally likely to fail and that the location of failures are uncorrelated.

In this chapter we will evaluate the disconnection probability in a family of regular graphs. The measure of *Network Resilience* is introduced as a probabilistic alternative to the static measure of

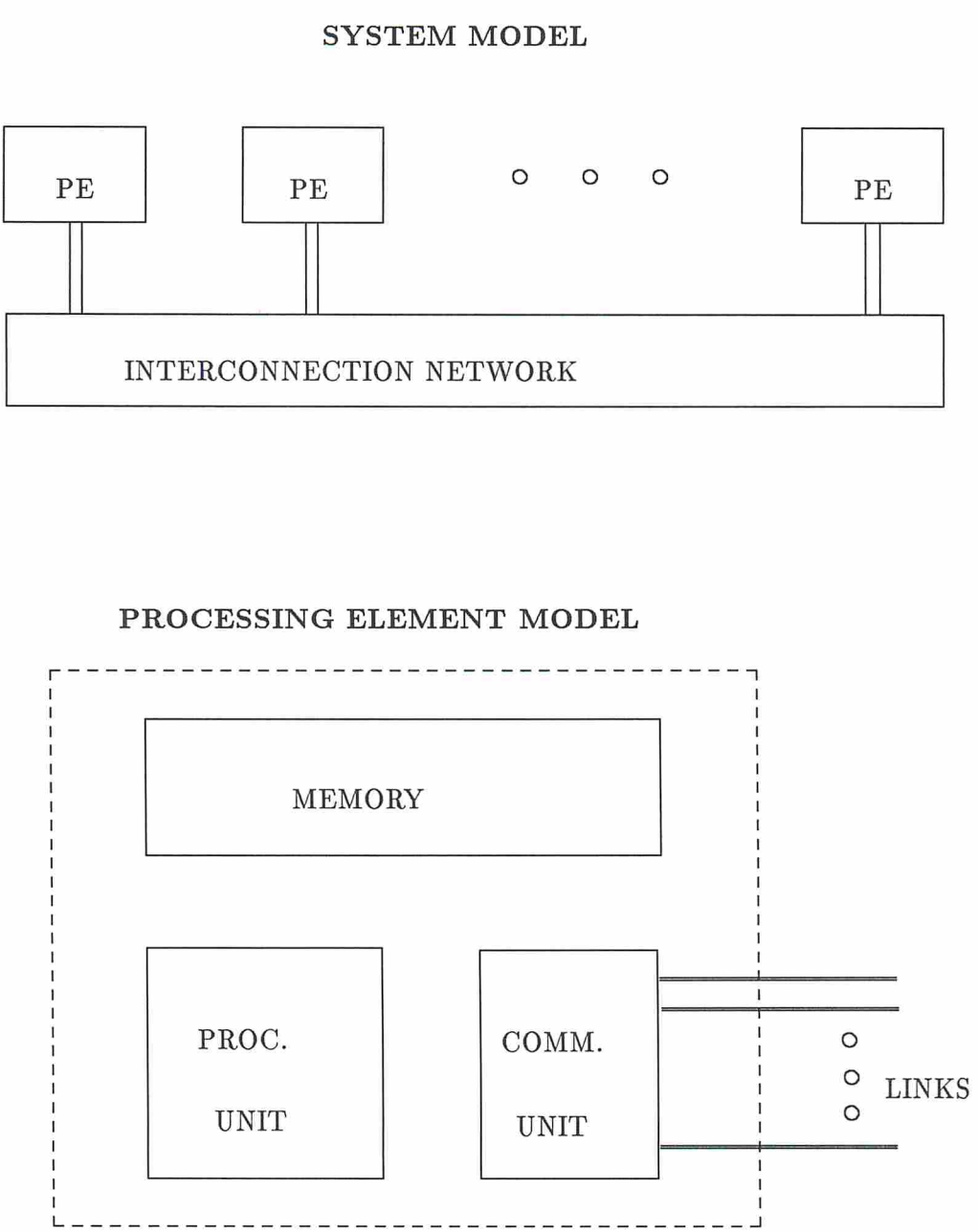


Figure 3.1: System and Processor Models

Network Fault-Tolerance. The obtained results are shown to be extensible, under certain conditions, to non-regular graphs. Finally, we demonstrate the analogy between node and link failures with respect to graph partitioning and extend the obtained results to include link failures.

3.1 Disconnection Probability

In this section we propose a combination of analytical and experimental approaches to the evaluation of the disconnection probability in a given network topology. Our model is a homogeneous, non-reconfigurable multiple processor system based on a class of n-regular graph network topology. An n-regular graph is a graph where the degree of all nodes is constant and equal to n. We assume that the loss or failure of a node implies the loss of all its connecting links.

The topological effects of a link or node failure are shown in Figure 3.2. Note that from the standpoint of node A, the loss of node B (Figure 3.2 b) or the link [AB] (Figure 3.2 c) have the same effects and the same potential of disconnecting node A from the rest of the system. However, the loss of node B has the same effect on three other nodes besides node A, while the loss of the [AB] link affects nodes A and B only.

After introducing the notations and formal definitions we analyze the initial disconnection probability and show that the case case of the single node disconnection is the most probable. These results are confirmed experimentally using a Monte-Carlo simulation. Based on these results, we propose an analytical approximation to the disconnection probability. Analytical and simulation results are compared and contrasted.

3.1.1 Notations and Definitions

Let $G(N, E)$ be an n-regular graph, with N nodes and E edges. A K-cluster is any connected subset of K nodes in $G(N, E)$. V_K is the number of neighbor nodes to a K-cluster and S_K is the number of K-clusters in $G(N, E)$.

Definition 3.1 *A system is in a disconnected state if and only if there exists a cluster of size K that is disconnected from the system and $K \geq 1$.*

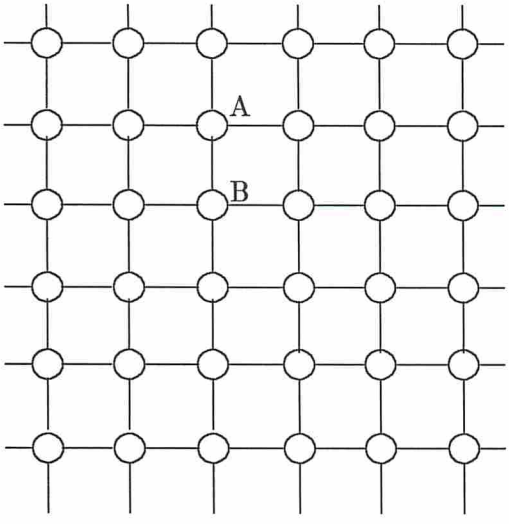
The analytical evaluation of the disconnection probability assumes a graph topology that satisfies the following two properties:

Property 3.1 *Let κ be the size of the mincut set of an n-regular graph $G(V, E)$, then $\kappa = n$.*

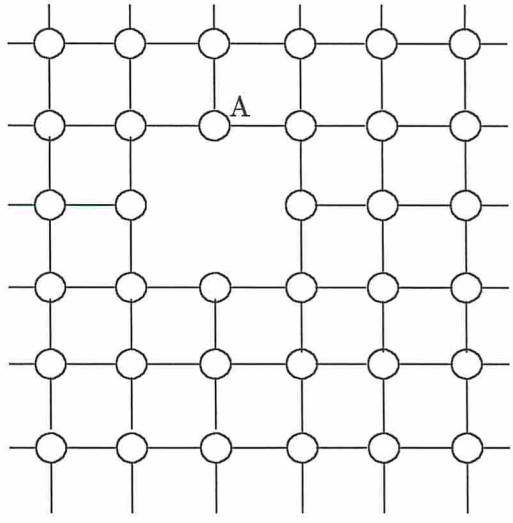
Property 3.2 *Given a K-cluster in $G(V, E)$, $1 < K < N/2 \Rightarrow V_K > n$.*

These definitions state that the size of the mincut set is equal to the node degree n, and the number of neighbors of a K-cluster is larger than n for $K > 1$. Therefore, the ring which is a regular graph, does not satisfy the stated properties. In this study, we consider, as examples, three popular topologies that satisfy these properties:

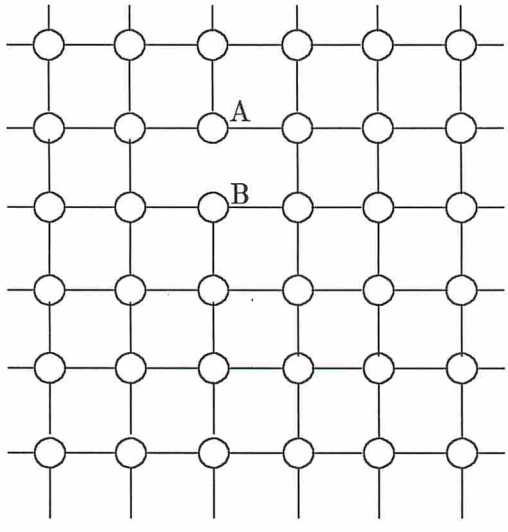
- The *cube-connected-cycles* topology proposed by Preparata and Vuillemin in [PV81]. This graph is based on a binary k-cube topology where each vertex is formed by a ring of nodes connected as a cycle, some nodes are also connected to nodes on neighboring vertices corresponding to a dimension of the binary cube. In the general case, the cube-connected cycles topology is **not** regular. In the examples we will consider a modified, but regular, version



a- Initial Condition



b- Node Failure



c- Link Failure

Figure 3.2: Node and Link Failures

of the cube-connected cycles where the number of nodes $N = k2^k$ and therefore each vertex consists of a ring with exactly k nodes each having 3 neighbors. The graph has a constant connectivity $n = 3$.

- The *torus* or wrap-around mesh such as the topology of the ILLIAC-IV multiprocessor system. The graph forms a surface where each node has a North, South, East and West neighbors. It is a regular topology with constant connectivity $n = 4$.
- The *binary k-cube* is a topology commonly known as the hypercube and used in several commercial systems [Sei85]. The number of nodes $N = 2^k$ where k is the *dimension* of the cube. Each node has k neighbor, one along each dimension of the cube. It is a variable connectivity graph with $n = \log_2 N$.

We define the following probabilities:

Definition 3.2 $P(i)$ = **Prob** *[the system is disconnected* exactly *after the i^{th} failure]*

Definition 3.3 $Q(i)$ = **Prob** *[a disconnected graph with $(N - i)$ nodes | a connected graph with $(N - i + 1)$ nodes and one node removal]*

Definition 3.4 $Q_K(i)$ = **Prob** *[a disconnected cluster of size K in a graph with $(N - i)$ nodes | a connected graph with $(N - i + 1)$ nodes and one node removal]*

Note that $Q(i)$ and $Q_K(i)$ are static conditional probabilities that do not take into account the evolution of the system. They are the probability of a disconnection event happening as the system goes from $(i-1)$ to i failures. $P(i)$, on the other hand, is a dynamic probability distribution function of the number of failures i. It implies that no disconnection occurred prior to the i^{th} failure. From these definitions, we can obtain the following relation:

$$P(i) = Q(i) \prod_{j=1}^{i-1} (1 - Q(j)) \tag{3.1.1}$$

This relation states that the probability of a disconnection event at the i^{th} failure is the product of the probability of no disconnection prior to that failure and the conditional probability of a disconnection at exactly the i^{th} failure. $Q(i)$ can also be written as:

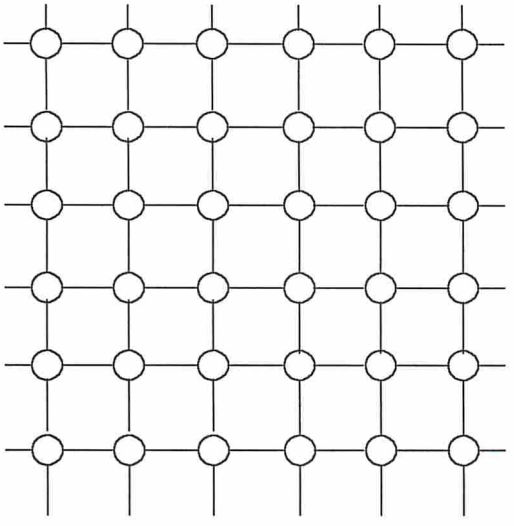
$$Q(i) = \sum_{K=1}^{M} Q_K(i) \tag{3.1.2}$$

Where M is the maximum possible value of K in a given graph topology.

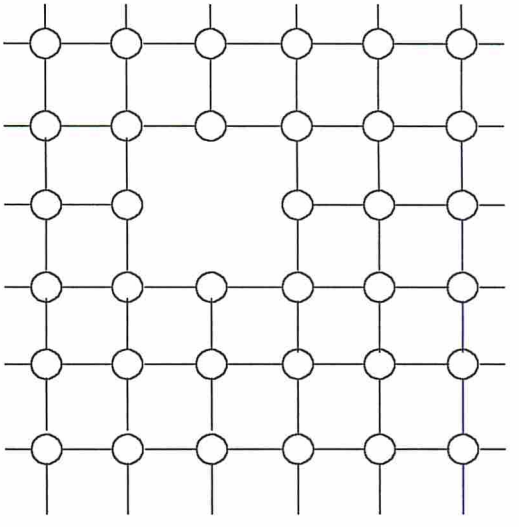
3.1.2 Initial Disconnection Probability

A state of disconnection is depicted in Figure 3.3 for $K = 1$ and $K = 2$ in a torus topology. When a node fails (3.3b), its corresponding links are also lost. Multiple failures result in the disconnection of a cluster of nodes (3.3c and d).

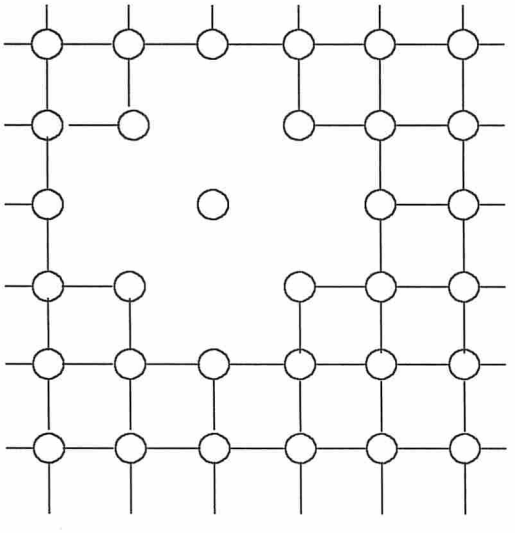
Addressing the problem of the disconnection probability in the general case, for all possible values of K, is practically impossible. The number of possible combinations of K connected nodes



a- Initial Condition

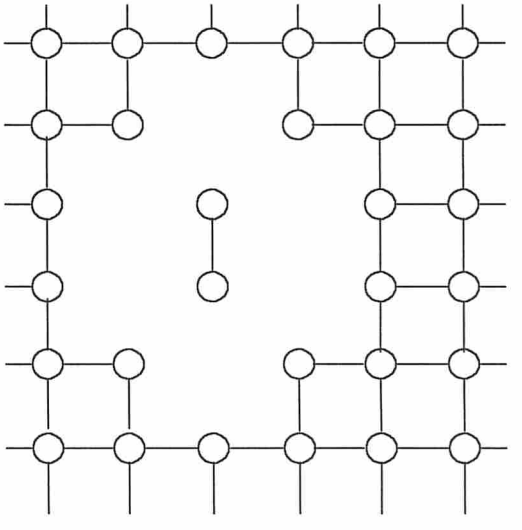


b- Node Failure



c- Single Node Disconnection

($K = 1$)



d- Two Nodes Disconnection

($K = 2$)

Figure 3.3: Node failures and disconnection

in a given network topology can grow exponentially with K. We therefore choose to address first a simpler version of the problem, that is the probability of disconnecting a *single* node in the system.

Since in a regular topology every node has exactly n neighbors, we can state that:

$$P(i) = 0 \qquad for \;\; i < n$$

and

$$P(n) = Q_1(n)$$

For a single node to be disconnected, at least n nodes must fail. For the disconnection to occur at the $i = n$ failure, all the n neighboring nodes must fail. In a system with N nodes, there are $\binom{N}{n}$ ways this can happen, among N possible nodes to be disconnected. Therefore:

$$Q_1(n) = \frac{N}{\binom{N}{n}} \tag{3.1.3}$$

The above result can be extended to the conditional probability of disconnection for K nodes cluster and for $i = V_K$, as:

$$Q_K(V_K) = \frac{S_K}{\binom{N}{V_k}} \tag{3.1.4}$$

Conjecture 3.1 *For large N and $N >> n$, $Q_1(i) >> (Q(i) - Q_K(i))$ or $Q(i) \approx Q_1(i)$*

This conjecture states that in large networks, the single node disconnection event ($K = 1$) is much more probable than any K-cluster disconnection where $K > 1$. This conjecture can be explained intuitively by considering that as the cluster size K increases, the number of neighbors increases too. Therefore the likelihood of that many nodes failing in a pattern that would result in a disconnected cluster decreases.

Proof: The conjecture can be proved by an analysis of Equations 3.1.4 and 3.1.3. For large networks and small n, *i.e.* $N \gg n$, the following approximation is justified:

$$\binom{N}{n} \approx \frac{N^n}{n!}$$

Since V_K is $O(n)$ and S_K is $O(N)$, therefore:

$$\frac{Q_1(n)}{Q_K(V_K)} \approx \frac{N^{(V_K - n)}}{(V_K - n)!}$$

From Sterling's formula we can derive the following approximation:

$$m! \approx \frac{m^m}{e^m}$$

it follows that:

$$\frac{Q_1(n)}{Q_K(V_K)} \approx \left(\frac{Ne}{(V_K - n)}\right)^{(V_K - n)}$$

In Section 3.4 we demonstrate that $Q_1(i+1) > Q_1(i)$ for $i > n$. It follows that:

$$Q_1(V_K) >> Q_K(V_K) \quad for \quad N >> n$$

Since we are considering failures occurring in sequence, this means that the probability of a single node disconnection at any $i < V_K$ is much larger than that of a cluster of size K at $i = V_K$. Therefore the single node disconnection probability is the dominant factor in $Q(i)$ and $Q(i) \approx Q_1(i)$.□

Following are two practical examples that demonstrate the above reasoning:

Example 1: Consider a torus topology ($n = 4$) with N nodes and the case of two nodes clusters ($K = 2$). In a torus, the number of neighbors to a two nodes cluster is $V_2 = 6$ and there are $S_2 = 2N$ such clusters in the network. Therefore the conditional probabilities of a single node disconnection at $i = n = 4$ failures ($Q_1(4)$) and of a two nodes disconnection at $i = V_2 = 6$ ($Q_2(6)$) are given respectively as:

$$Q_1(4) = \frac{N}{\binom{N}{4}} = \frac{24}{(N-1)(N-2)(N-3)}$$

$$Q_2(6) = \frac{2N}{\binom{N}{6}} = \frac{1440}{(N-1)(N-2)(N-3)(N-4)(N-5)}$$

For a system size of $N = 256$ nodes, we have:

$$Q_1(4) = 1054\ Q_2(6)$$

In section 3.1.4 we prove that $Q_1(i+1) > Q_1(i)$ for $i > n$, therefore:

$$Q_1(6) > 1054\ Q_2(6)$$

Example 2: A hypercube topology with $N = 2^n$ nodes, and $n = 8$. $S_2 = nN/2 = 2048$ and $V_K = (n - log_2 K)K$ therefore $V_2 = 14$.

$$Q_1(8) = \frac{256}{\binom{256}{8}} = 625\ 10^{-15}$$

$$Q_2(14) = \frac{2048}{\binom{256}{8}} = 0.2\ 10^{-18}$$

Hence:

$$Q_1(14) > 3\ 10^6\ Q_2(14)$$

These examples show that when the disconnection of a two nodes cluster is possible (at $i = 2(n-1)$) the probability of a prior single node disconnection event is a thousand times larger in the torus case and three million times larger in the hypercube case. The above two examples are not a proof, but a demonstration of the rationale behind the proposed conjecture for two common network topologies. In the following section an experimental approach based on a probabilistic, Monte-Carlo, simulation is presented that verifies the proposed conjecture.

	Cube-Connected Cycles, N=					
K	24	64	160	384	896	2048
1	**25.2**	**52.2**	**71.3**	**78.6**	**84.0**	**86.1**
2	18.6	13.0	15.3	14.96	12.2	12.0
3	11.0	10.0	5.0	3.78	2.5	1.5
4	7.0	6.5	3.0	1.22	0.9	0.4
	Torus, N=					
K	16	64	100	256	400	1024
1	**69.93**	**66.37**	**69.9**	**81.05**	**83.8**	**90.85**
2	18.1	10.8	11.07	8.4	8.15	4.05
3	8.13	6.17	6.2	4.55	3.95	2.6
4	1.45	4.01	3.6	1.75	1.05	0.75
	Binary Cube, N=					
K	16	64	128	256	512	1024
1	**67.75**	**77.25**	**89.1**	**92.4**	**95.1**	**97.3**
2	11.25	9.4	6.65	5.4	4.1	2.4
3	8.05	3.2	1.85	0.95	0.6	0.3
4	5.95	2.15	0.65	0.45	0.2	0.0

Table 3.1: Frequencies of Disconnection

3.1.3 Monte-Carlo Simulation

The objective of the simulation is to measure the values of $P(i)$ for different values of N on the three example topologies described in section 3.1.1: the cube-connected cycles, the torus and the binary k-cube.

The Monte-Carlo simulation algorithm used to obtain these results is described in Figure 3.4.

We first present the frequency of occurrence of a disconnections of different sizes clusters for the three topologies under considerations. This measure is defined as follows:

Definition 3.5 $F_{graph}(K)$ = **Prob***[the size of the disconnected cluster is* K | *a disconnection occurred]*

Given that a disconnection occurred in a given *graph*, $F_{graph}(K)$ is the probability that the disconnected component is a cluster of size K. F_t, F_{bc} and F_{ccc} are the values of F_{graph} for the torus, binary cube and cube-connected cycles topologies respectively. These values are shown in Table 3.1 for $K = 1, 2, 3$ *and* 4, as obtained from the Monte-Carlo simulation.

From these results, we can draw the following conclusions:

- Except for the cube-connected cycles with $N = 24$, in all three cases, and for all values of N, the proposed conjecture is verified in that the frequency of a single node disconnection, $F_{graph}(1)$, is larger than 50%. Furthermore, $F_{graph}(1)$ increases with increasing N independently of the connectivity.

- Comparing the results for the three topologies, one can observe that the dominance of the single node disconnection increases with an increasing n. For example, for $N = 64$, one can observe that: $F_{bc}(1) > F_t(1) > F_{ccc}(1)$ and $F_{bc}(K) < F_t(K) < F_{ccc}(K)$ for all $K > 1$

These effects can be explained by recalling the expression for $Q_K(V_K)$. Although the numerator, S_K, increases with an increased connectivity n, the increase in the denominator, $\begin{pmatrix} N \\ V_K \end{pmatrix}$, is the dominant factor. Therefore, for large system sizes, the dominance of the single node disconnection probability on the overall disconnection probability can be stated by the following approximation:

$$Q_1(i) >> Q(i) - Q_1(i) \Longrightarrow Q(i) \approx Q_1(i)$$

which states that the conditional disconnection probability can be approximated by the single node disconnection probability. The above results do not give an indication on the value of $P(i)$ itself, only on its composition. In the following section we propose an approximation to $P(i)$ based on $Q(i) \approx Q_1(i)$ for large values of N.

3.1.4 Single-Node Disconnection Approximation

In this section we propose an approximate expression for $P(i)$ based on the results shown in the previous section. In this effort, the objective is **not** to provide an exact value for $P(i)$, but rather an indication of its order of magnitude in an analytical way.

The proposed approximation, for large N, is expressed as follows:

$$Q_1(i) >> Q(i) - Q_1(i) \quad or \quad Q_1(i) >> Q_K(i) \quad \forall K > 1 \tag{3.1.5}$$

Therefore we can approximate $P(i)$ as:

$$P(i) \approx P_1(i) = Q_1(i) \prod_{j=1}^{i-1} (1 - Q_1(j)) \tag{3.1.6}$$

In order to evaluate $P(i)$ we need an expression for $Q_1(i > n)$. It is provided by the following theorem:

Theorem 3.1 *The conditional probability of disconnecting a single node after i failures, where* $n < i < 2n - 1$ *is given by:*

$$Q_1(i) = \frac{nN \begin{pmatrix} N-n-1 \\ i-n \end{pmatrix}}{(N-i+1) \begin{pmatrix} N \\ i-1 \end{pmatrix}} \; for \; n < i < 2n-1; \tag{3.1.7}$$

Proof: Keeping in mind that no disconnection happened at the $(i-1)^{st}$ failure, the probability that one occurs at the i^{th} failure is the probability that some node had all but one of its neighbors failed *and* that that neighbor was the i^{th} failure. $\begin{pmatrix} N \\ i-1 \end{pmatrix}$ represents the possible combinations of $i-1$ failures among N nodes. $\begin{pmatrix} n \\ n-1 \end{pmatrix} = n$ is the possible combinations of $n-1$ failed neighbors among n neighbors. $\begin{pmatrix} N-n-1 \\ i-n \end{pmatrix}$ is the combination of the remaining $i-n$ failures

```
for (j = 0; j < Number-of-iterations; j++) {
        Build a graph;
        for (i = 1; i < N; i++){
                • choose a node at random from the remaining (N − i);
                • remove that node and all its links from the graph;
                • traverse the graph recording the number
                        and size of connected components;
                • if (disconnection){
                        record i;
                        record size of cluster;
                        exit;}
        Report data;}
}
```

Figure 3.4: Monte-Carlo Simulation Algorithm

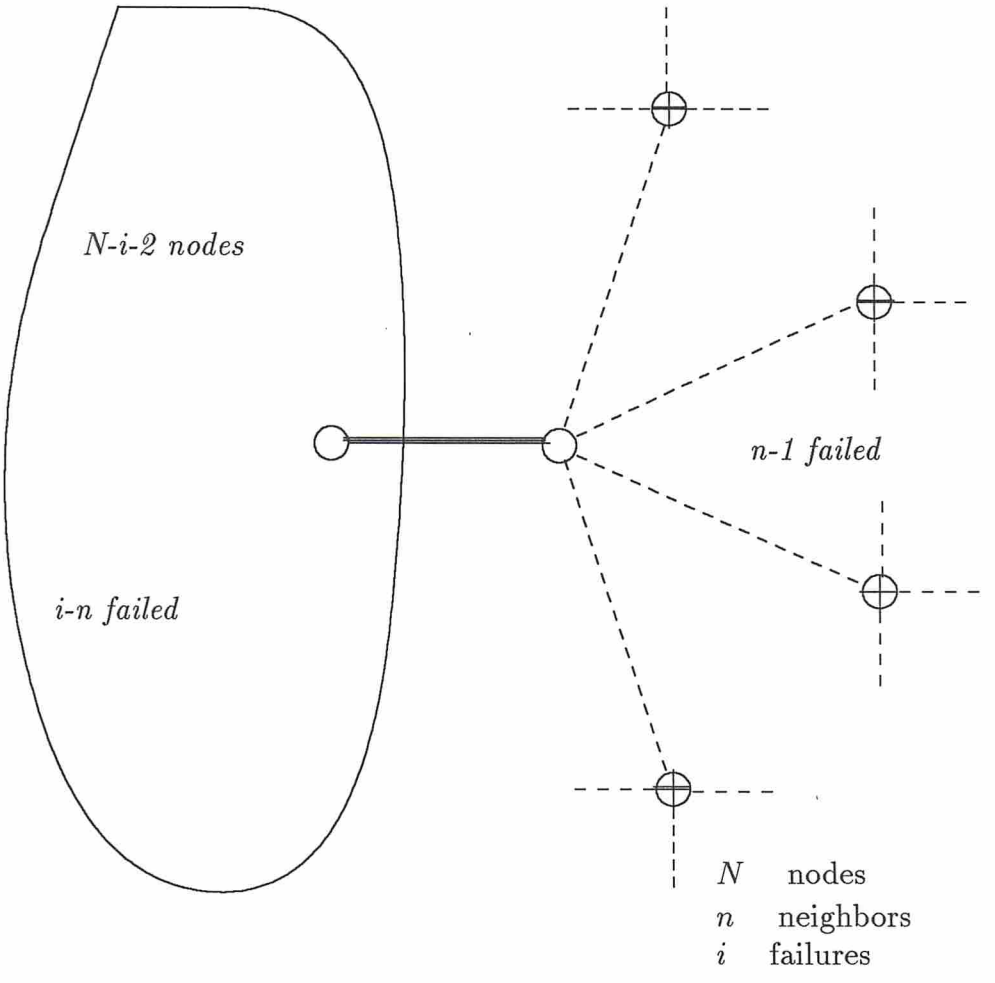


Figure 3.5: Disconnection when $i > n$

in the rest of the system, where N is the number of nodes which can be isolated, and $\frac{1}{N-i+1}$ is the probability that the last remaining neighbor fails. □ Equation 3.1.7 corresponds to the single node disconnection probability when more than n nodes have failed. For $i \geq 2n-1$ it is possible to have two ore more single node disconnections. However, the probability of multiple single node disconnections is of the same order as that of a cluster disconnection where $K > 1$. Therefore, using the approximation of Equation 3.1.5, we can extend the range of i in Equation 3.1.7 to $i > n$. Simplifying Equation 3.1.7 we obtain:

$$Q_1(i > n) = \frac{n(N-n-1)!(i-1)!(N-i)}{(i-n)!(N-1)!} \tag{3.1.8}$$

From Equation 3.1.8 we can derive:

$$\frac{Q_1(i+1)}{Q_1(i)} \frac{i}{i+1-n} \frac{N-i-1}{N-i} \approx \frac{i}{i+1-n}$$

This proves the relation $Q_1(i+1) > Q_1(i)$ for $i > n$.

The single node disconnection approximation can therefore be summarized as:

$$P(i) = 0 \quad for \;\; i < n$$

$$P(n) = Q_1(n) = \frac{N}{\binom{N}{n}}$$

$$P(i) = Q(i) \prod_{j=1}^{i-1} (1 - Q(j))$$

and

$$Q(i) \approx Q_1(i) = \frac{n(N-n-1)!}{(N-1)!} \frac{(i-n)!}{(i-1)!(N-i)} \quad for \;\; i > n$$

In the following section we present a comparison of the disconnection probability values obtained with this approximate analytical model and the Monte-Carlo simulation described in the previous section.

3.1.5 Analytical and Simulation Results

Using the same simulation approach described in Section 3.3, we obtained the values of $P(i)$ for the three topologies and for different values of N.

Figure 3.7 shows the values of $P(i)$ for a torus with 64 nodes as a function of i. The two curves show the simulation and analytical values. The maximum difference in value, at the peak, is 25%. Figure 3.7 shows the same curve for the case of a binary cube, the maximum difference being 20%. Figures 3.9 and 3.8 show similar plots for $N = 256$. One can notice that the curves for the binary cube cases are narrower and peak around $i = N/2$, while they are wider for the torus cases with a peak at $i < N/2$.

Table 3.2 summarizes the comparison between the simulation results and those obtained with the analytic approximation. These tables show the peak value of the overall disconnection probability distribution function $P(i)$, denoted by P_{max}, and the corresponding value of the number of node failures, i_{peak}. The objective in this comparison is to show the order of magnitude of $P(i)$ and the corresponding *range* of values of i where the peak value of the disconnection probability occurs.

From this table one can observe the following:

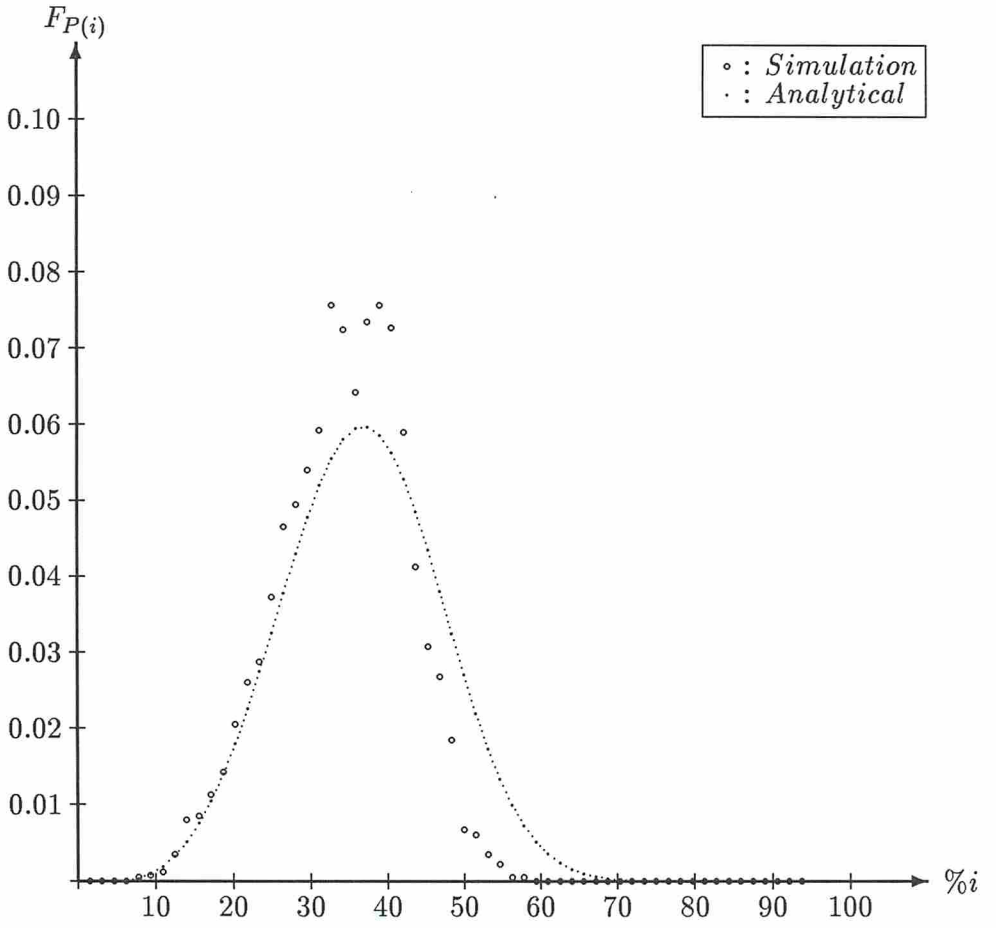


Figure 3.6: Probability of Disconnection, Torus, $N = 64$

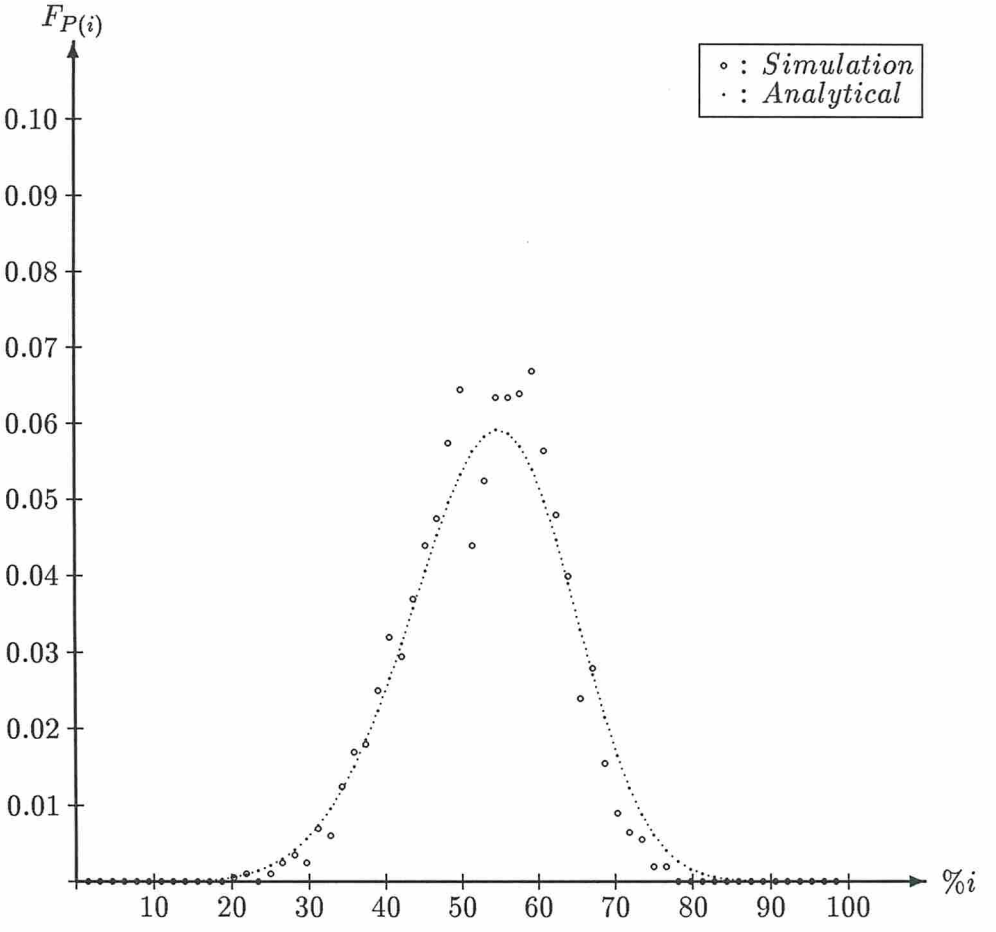


Figure 3.7: Probability of Disconnection, Binary Cube, $N = 64$

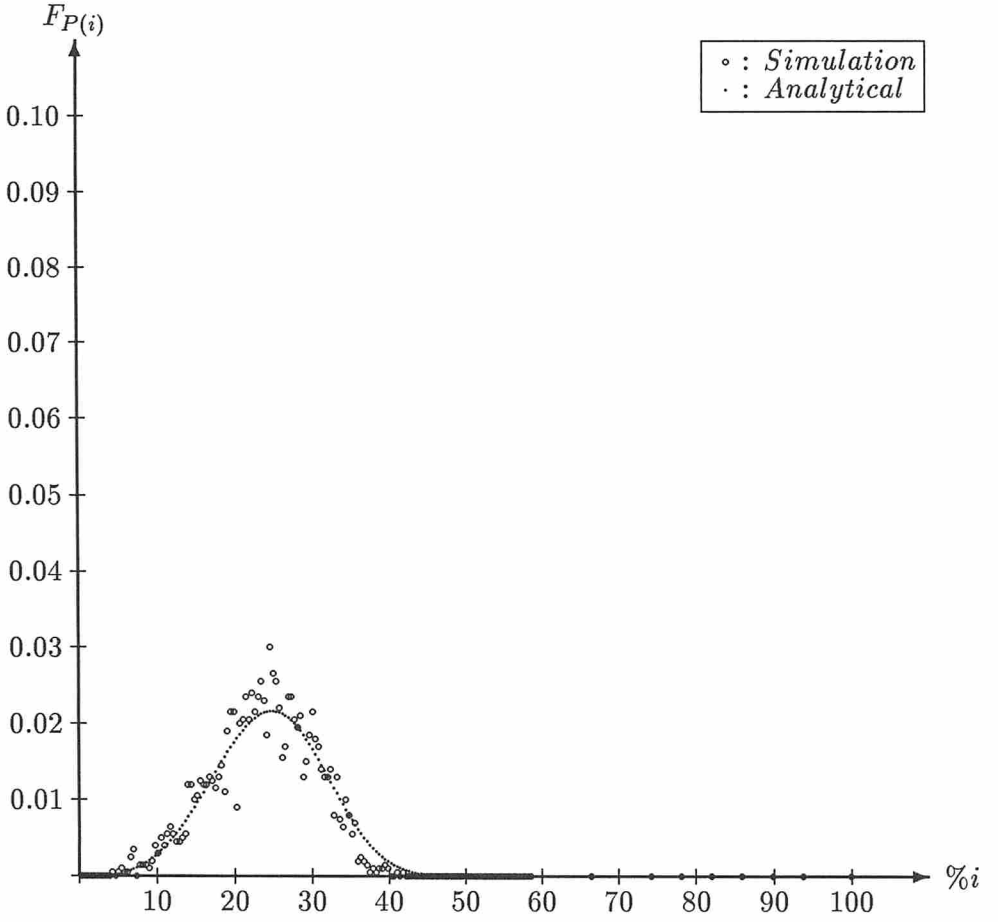


Figure 3.8: Probability of Disconnection, Torus, $N = 256$

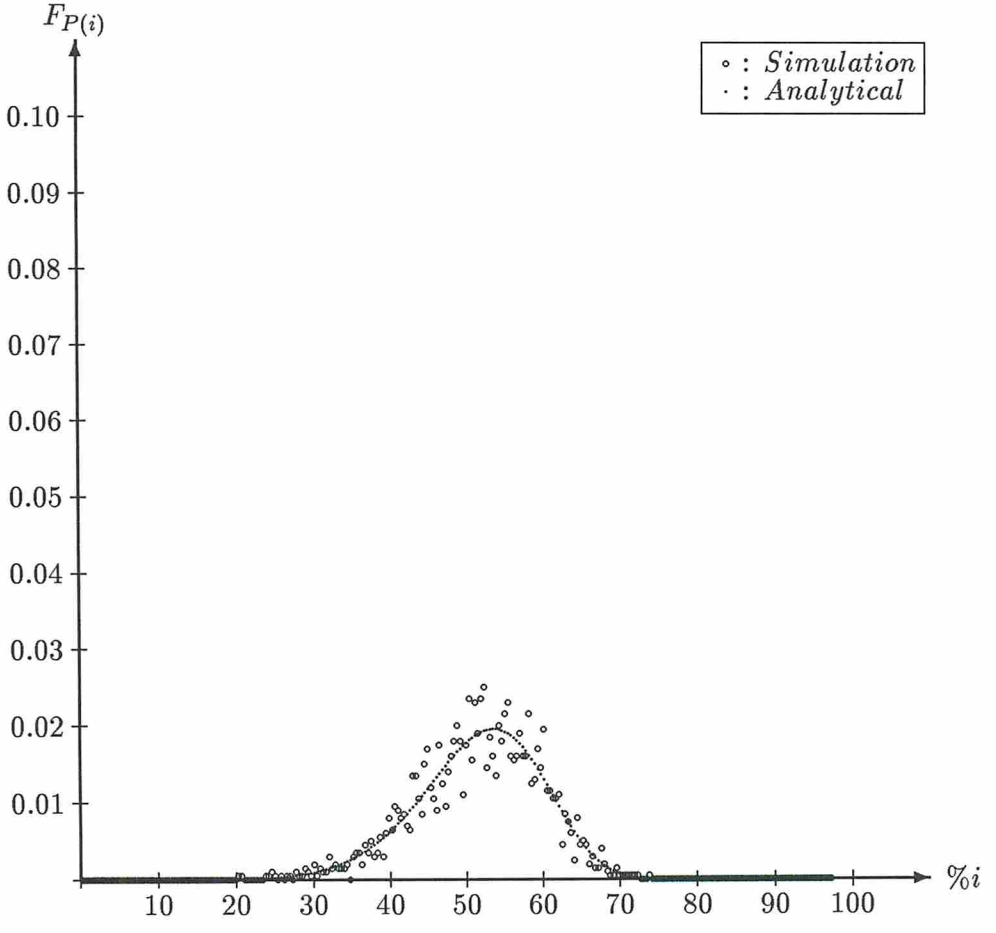


Figure 3.9: Probability of Disconnection, Binary Cube, $N = 256$

		Cube-Connected Cycles					
N		24	64	160	384	896	2048
Simulation	P_{max}	0.296	0.095	0.044	0.028	0.0187	0.016
	i_{peak}	6	15	28	50	84	114
Analytical	P_{max}	0.132	0.068	0.0375	0.021	0.012	0.007
	i_{peak}	9	16	28	48	84	144
		Torus					
N		16	64	100	256	400	1024
Simulation	P_{max}	0.2066	0.0775	0.058	0.03	0.0215	0.0105
	i_{peak}	9	25	32	63	88	190
Analytical	P_{max}	0.1765	0.0597	0.043	0.0216	0.0156	0.00785
	i_{peak}	9	24	32	64	88	175
		Binary Cube					
N		16	64	128	256	512	1024
Simulation	P_{max}	0.2122	0.069	0.0375	0.025	0.0165	0.0117
	i_{peak}	9	38	67	142	284	575
Analytical	P_{max}	0.1765	0.0593	0.0342	0.0195	0.0110	0.0062
	i_{peak}	9	35	69	137	272	541

Table 3.2: P_{max} and i_{peak}

- A very close correlation in the value of i_{peak} between the simulation results and the analytical model. The variation in the value of P_{max} is due to the approximation in the analytical model as well as to the statistical error in the simulation.
- In the cube-connected and mesh topologies, $i < N/2$ while $i_{peak} > N/2$ in a hypercube topology. For example, with $N = 1024$ we have $i \approx 200$ for the mesh and $i \approx 550$ for the hypercube. This shows that higher graph connectivity allows the network to sustain a larger number of failures before disconnection.
- The value of P_{max} falls within the same order of magnitude for all three topologies and for a given range of values of N. For example, $P_{max} \approx 20\%$ for $N = 16$ and $P_{max} \approx 2.5\%$ for $N = 256$.

These results show that the number of nodes in the system, N, is the dominant factor in determining the maximum value of the disconnection probability, P_{max}. On the other hand, the node connectivity, n, determine the range of values of i_{peak}. In other words: the *magnitude* of the disconnection probability id determined by the system size (N), while the expected number of failures that can be sustained before a disconnection occurs is determined by both the system size and the node connectivity.

The coverage factor, in a gracefully degradable system, is the probability of successful recovery. We have shown that a disconnection in the network will prevent a successful recovery mechanism. Therefore, the probability of *no* disconnection is a multiplicative coefficient in the expression of the coverage factor. In other words, the coverage factor at the i^{th} failure can be expressed as:

$$c_i = (1 - P(i))c'_i$$

	Cube-Connected Cycles					
N	24	64	160	384	896	2048
Simulation	2	4	7	12	20	40
Analytical	2	4	7	12	21	35
	Torus					
N	16	64	100	256	400	1024
Simulation	4	8	11	19	28	55
Analytical	4	8	11	21	30	59
	Binary Cube					
N	16	64	128	256	512	1024
Simulation	4	18	33	75	159	336
Analytical	4	17	36	77	161	337

Table 3.3: Network Resilience for $p = 0.01$

where c_i' is the coverage factor in a system where no disconnection can occur as for example in a fully connected graph topology. The range of values of P_{max} shown in Table 3.2 is very high compared to any acceptable value of the coverage factor. The question therefore becomes: What is the number of nodes that can be allowed to fail with a reasonably low probability of disconnection? This evaluation is the topic of the next section.

3.2 Network Resilience

Network Resilience is introduced as measure of the expected number of nodes failures a system graph can sustain with a *reasonable* probability of no network disconnection. The reasonable probability is determined by a *certainty factor* $(1 - p)$. Network resilience is therefore defined as:

$$NR(p) = \sum_{i=1}^{NR} P(i) \leq p \tag{3.2.9}$$

NR is therefore the cumulative distribution function of $P(i)$. In a similar fashion, we define the measure of *Relative Network Resilience*, RNR, as:

$$RNR(p) = \frac{NR(p)}{N} \tag{3.2.10}$$

While $NR(p)$ measure the absolute number of nodes, $RNR(p)$ measures the percentage of nodes and gives therefore an indication of the scalability of various network topology with respect to network fault-tolerance.

Figure 3.10 shows the plots of $NR(p)$ for all three topologies for $p = 0.01 = 1\%$. Figure 3.11 shows similar plots of $RNR(0.01)$. The numerical values are displayed in Table 3.2.

It is interesting to notice that for increasing N the values of $RNR(p)$ decreases in the case of the cube-connected cycles and mesh while it increases for the hypercube. For the range of values shown in Table 3.2 for the respective topologies, this ratio goes from 8% ($N = 24$) to 1.9% ($N = 2048$)

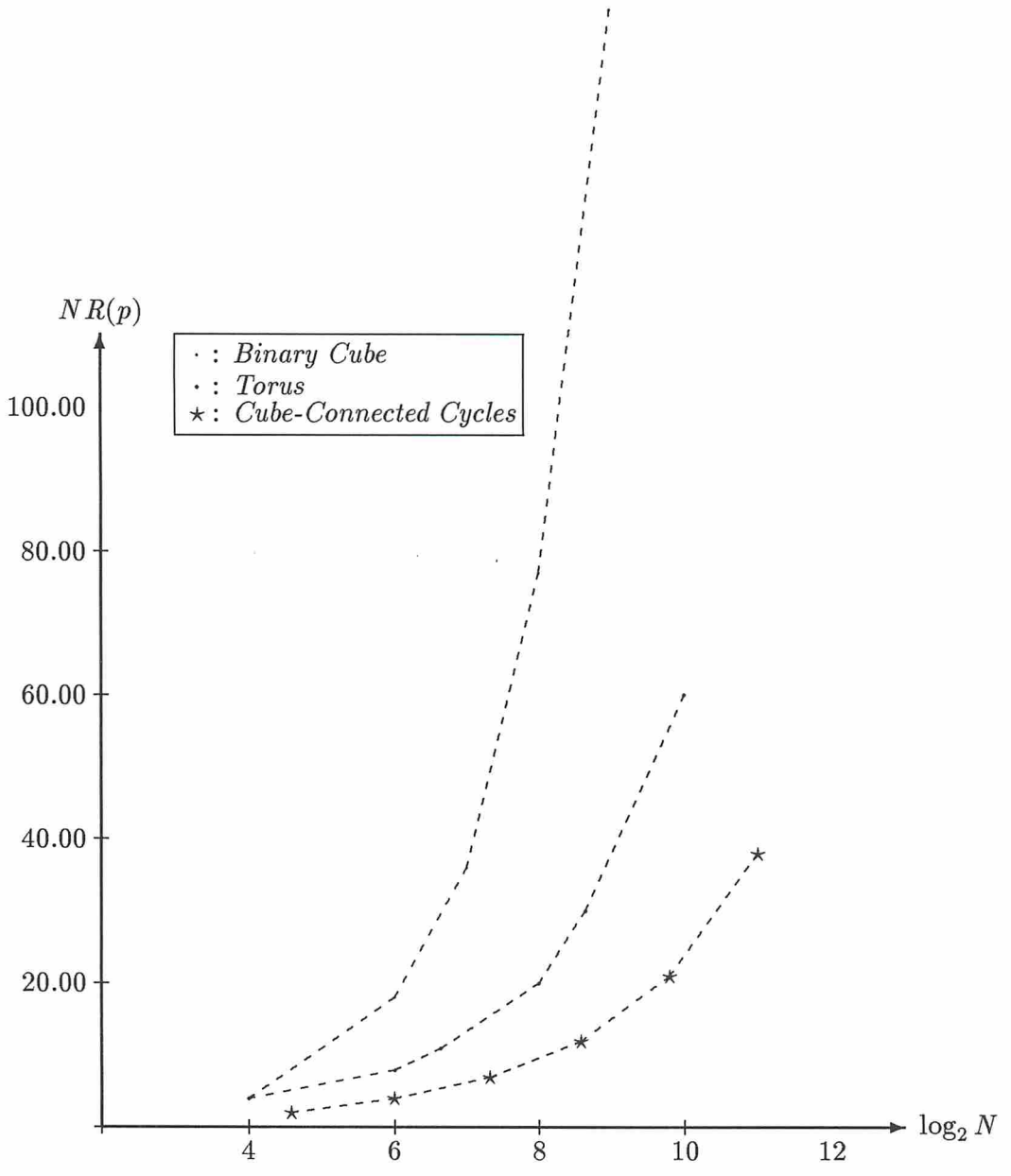


Figure 3.10: Network Resilience, $NR(p)$, for $p = 0.01$

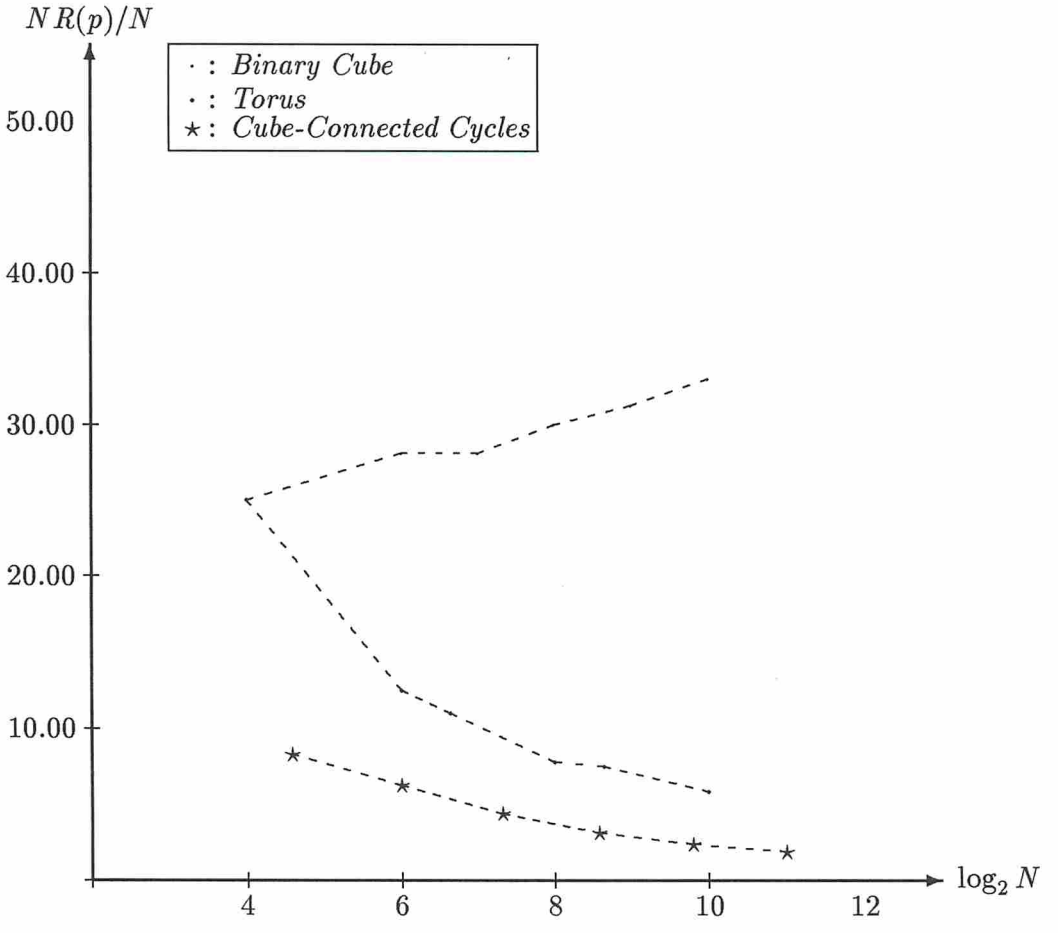


Figure 3.11: Relative Resilience, $NR(p)/N$, for $p = 0.01$

for the cube-connected cycles, from 25% ($N = 16$) to 5.6% ($N = 1024$) for the mesh and from 25% ($N = 16$) to 33% ($N = 1024$) for the hypercube.

From these observations we can conclude:

- The value of $NR(p)$ increases with increasing N. Therefore, larger systems can allow a larger number of nodes to fail before the cumulative probability of disconnection reaches a given level.
- The value of $RNR(p)$ decreases with increasing N for constant constant connectivity graphs such as the torus or the cube-connected-cycles. Therefore, for a constant n, the number of degradation states is a decreasing fraction of the number of nodes as N increases.
- For networks where the connectivity n is an increasing function of the number of nodes N, as in the binary cube case, an increase in N increases both $NR(p)$ and $RNR(p)$.

The comparison between analytical and simulation values in Table 3.2 indicates that the adopted analytical model yield very close values of $NR(p)$ and $RNR(p)$. While network fault-tolerance is a static measure that depends exclusively on n and therefore on the topology alone, network resilience is a probabilistic measure that depends on both N and n. It can be tuned by selecting appropriate values of the certainty factor p. In fact for the trivial and conservative case where $p = 0$ the two measures of network fault-tolerance, as defined in [Pra85a], and network resilience as defined here are equivalent.

The notion of network resilience has implications on the number of degradation states that can be allowed in a degradable system. A large-scale gracefully degradable system can tolerate element failures while providing continued operations. The maximum number of node failures allowed in a multicomputer system is called the *degradation level*, denoted here as D. When the application does not pose any constraint on the minimum number of processing nodes necessary, the value of D is determined by the system fault-tolerant design. As the number of failures increases, the difficulty of system recovery increases, thereby decreasing the probability of a successful recovery. The degradation level, D, is therefore the number of failures up to which graceful degradation can be expected with reasonably high probability. After D failures, the recovery becomes difficult and the probability of success low, therefor the system is considered failed. In a distributed system that is not based on a fully connected graph, the value of D is constrained by the probability of network disconnection. By determining an acceptable value of the certainty factor p, network resilience can allow the determination of the number of degradation levels D.

3.3 Non-Regular Graphs

Non-regular graph topologies such as the array, offer the advantages of a constant connectivity that is independent of the system size and therefore of node modularity. Furthermore, the array, like the binary k-cube, is a *scalable* topology: An array (or binary k-cube) graph can be partitioned into several smaller arrays (cubes) having the same topology. This is not the case in graphs such as the torus where partitioning does not preserve the original network topology.

In this section we present a simulation analysis of the disconnection probability in an array topology. These results are then compared to those of a same size torus using the analytical model that was derived in section 3.1.

Table 3.4 shows the frequency of disconnection of K nodes cluster as a function of the array size. These results show that the frequency of a single node disconnection ($K = 1$) is still the dominant

	Array					
K	16	64	100	256	400	1024
1	55.7	49.7	56.8	65.7	72.6	76.0
2	12.6	10.9	10.9	9.9	9.9	15.0
3	13.5	10.4	8.7	7.7	5.8	3.0
4	8.6	5.2	6.1	3.7	3.8	3.0

Table 3.4: Frequencies of Disconnection, Array

N	16	64	100	256	400	1024
Torus	4	8	11	21	30	59
Array	1	2	3	10	16	23

Table 3.5: Comparison of Network Resilience, Torus and Array ($p = 0.01$)

factor. However, comparing the frequencies of disconnection in the array and torus topologies (Table 3.1), we can note that the frequency of single node disconnections in the array case is significantly smaller than in the torus case. This reduction is due to the *edge effect* introduced by the non-regularity of the graph. In an array topologies there are $4(\sqrt{N} - 1)$ nodes with degree 3 and 4 nodes with degree 2. The ratio of the number of edge nodes to the total number of nodes is $O(N^{\frac{1}{2}})$. It would be expected, therefore, that for very large system sizes the disconnection frequency in an array topology would approach that of a torus topology.

We note from Figure 3.12 that the edge effect results in an increase in the probability of disconnection as compared to that of a torus. Therefore, for a same fraction of failed nodes, the array is more likely to get disconnected than a torus.

Table 3.3 shows the effects of graph non-regularity on Network Resilience by comparing $NR(0.01)$ for the array and the torus graphs. These values show a very large decrease in network resilience. In fact, one can notice that for $N > 100$ the NR of an array is roughly half that of a same size torus. Therefore an array topology is twice as prone to network disconnection than a torus of the same size.

From this analysis, we can conclude that the problem of network disconnection is even more acute in the case of low degree non-regular graphs such as the array. Therefore, protection measures, such as backup nodes or links, must be provided in order to allow a larger number of graceful degradation states in systems based on such graph topologies. An alternative possibility would be to investigate the design of network topologies that minimize the ratio of edge nodes to total number of nodes thereby providing a higher resilience to network disconnection.

3.4 Link Failures

The analysis of sections 3.1 and 3.3 are based on the assumptions of node failures exclusively. In this section, we carry a similar analysis for the link failures, assuming no node failures.

In the multicomputer system model described in section 3.1, each node consists of a Processing Unit (PU) and a Communication Unit (CU). The failure of either of the PU or CU is tantamount to a node failure since no communication is possible to the outside world (in the case of a CU failure)

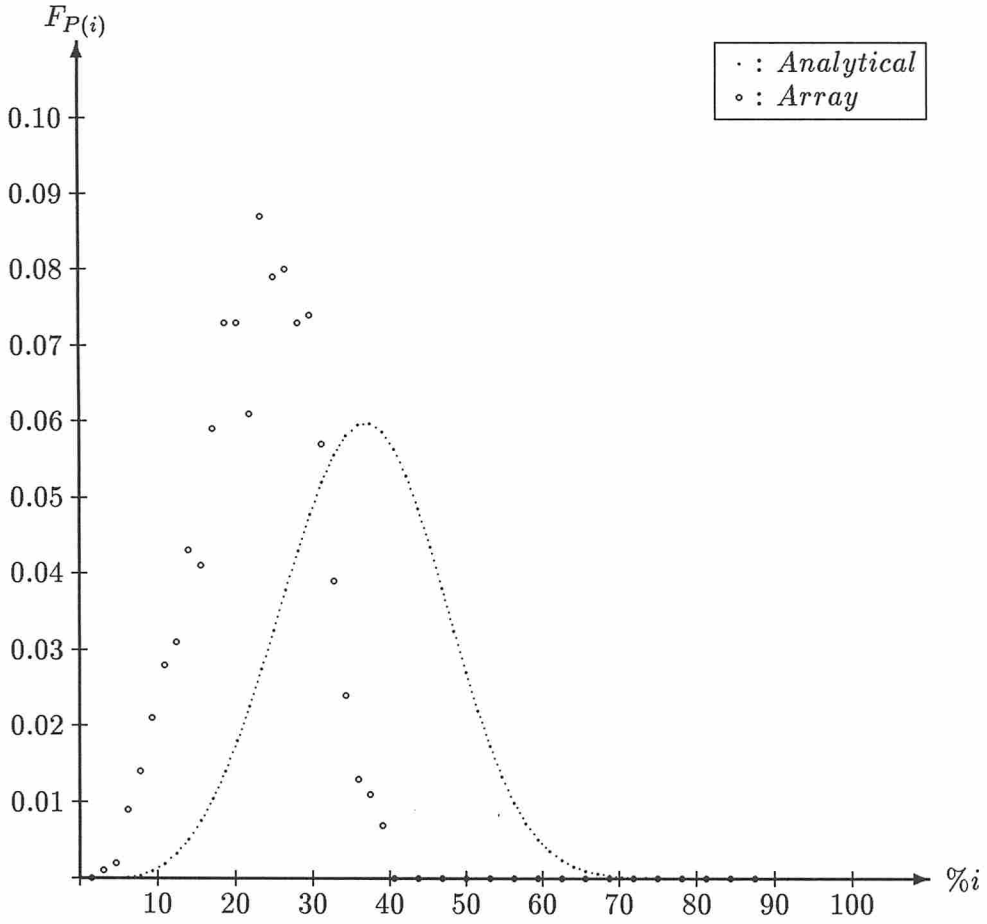


Figure 3.12: Probability of Disconnection, Array and Torus, $N = 64$

	Cube Size					
K	8	16	64	128	256	512
1	76.3	79.3	87.7	90.3	95.3	97.0
2	18.6	15.3	9.3	7.8	4.6	2.6
3	4.2	3.6	1.8	1.7	0.1	0.4
4	0.9	1.4	0.8	0.1	0.0	0.0

Table 3.6: Frequencies of Disconnection in a Binary Cube, Link Failures

or no useful work can be done in that node (in the case of a PU failure). The links connecting the nodes constitute the interconnection network.

Well known and simple fault-tolerance techniques exist for the protection against link failures. Most common among these are the various error detection and correction schemes. These are based either on the use of redundant bits for error-detection and error correction such as in SEC/DED codes, or error-detection bits with data retransmission (SED codes). Although these schemes can provide a very high level of protection, they are not fail-proof and therefore system failures can still be caused by single or multiple link failures. A particular case worth noting is when Very Large or Wafer Scale Integration technologies are used to implement several nodes on one chip. In such cases the silicon real-estate used by the interconnection network becomes a substantial fraction of the overall chip area. However, the same traditional protection techniques apply as well. Therefore, in all realistic cases, the expected link failure rate will be much lower than the expected node failure rate.

A Monte-Carlo simulation of link failure in binary n-cubes (Table 3.6) shows that the same behavior can be observed as in the node failure case. The most frequent disconnection is that of a single node. We can deduce, therefore, that for either link or node failures, protecting against the disconnection of a single node would provide a very high coverage.

Chapter 4

COMPUTATIONAL RELIABILITY

"... but I hope this restores your confidence in my reliability."

"I am sorry about this misunderstanding, Hal," replied Bowman, rather contritely.

Arthur C. Clarke
2001: A Space Odyssey

In Chapter 3 we have seen that an increase in system size has a favorable effect on the network disconnection probability by allowing a higher value of the Network Resilience. For low constant connectivity graphs, this value was shown to be a decreasing fraction of the system size as the number of nodes is increased. As the system size is increased, the performance and reliability of the system are influenced, respectively, by (1) an increase in the available computational power and (2) an increase in the expected rate of failure of the system. The objective of this chapter, therefore, is to propose a combined performance/reliability modeling and an analysis of gracefully degradable large-scale multicomputer systems. In this analysis, particular emphasis will be put on the following issues:

- the *scalability* of large-scale systems, by analyzing the effects of an increase in system size;
- the *quality of the recovery scheme*, by an analysis of the effects of the coverage factor.
- the *computational reliability* which is the probability of correct completion of a given computation, as function of both the system size and the coverage factor.

We present first the system and failure models on which the analysis in this chapter is based. This followed by an analysis of time-based measures such as Mean-Time-To-Failure and Mission-Time. Computation-based measures are then introduced, defined and analyzed. Finally, a discussion of these results and their implications on the design and performability of LSCSs is given.

4.1 System and Fault Models

The model under consideration is that of a large-scale, homogeneous multiprocessor. The computation is, initially, uniformly partitioned among N identical processing elements. The system is assumed to support graceful degradation. Upon the detected failure of a processor, its computational load is picked up by another processor or set of processors with near uniform load partitioning. Distributed fault-tolerant schemes are based on the algorithm proposed by Preparata

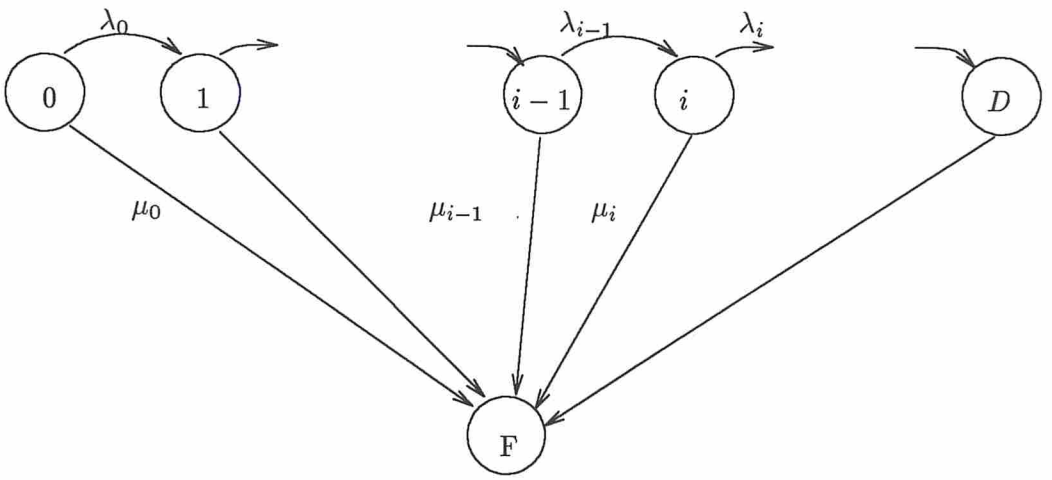


Figure 4.1: Markov model of failures

et al. [PMC67]. A detailed description and discussion of this algorithm can be found in [KR80]. It is based on the following three steps:

- fault detection
- fault isolation
- system reconfiguration and recovery

However, the ability of a system to gracefully degrade hinges on the *combined* success of these three steps. The failure to either detect, isolate or recover from a fault can result in a total system failure. The cumulative probability of success of these three steps is expressed by the *coverage factor* [BCJ*71]. In a distributed system, the recovery procedure relies on the communication among processors, and therefore on the system network being connected. Unless the system has a fully connected network topology, successive failures might result in a partitioning of the network. The probability of disconnection is, therefore, a parameter of the coverage factor as demonstrated in Section 3.1.2.

A common assumption of fault-tolerant schemes is that of no simultaneous multiple failures, in other words, that failures are sufficiently spaced to allow the recovery process to deal with one failure at a time. The occurrence of two or more failures in a short interval might lead to a failed recovery and therefore to a total system crash. For this reason, a *Global State Saving* (GSS) procedure is necessary at fixed intervals during the course of the computation to insure that the progress done by the computation is not lost. One of the goals of the present analysis is to evaluate the interval between successive GSS procedures: T_{gss}.

For the sake of simplicity, the analysis that follows will not take into consideration the time overhead incurred in recovering from a failure. Although this assumption is unrealistic, it is justifiable in an analysis of asymptotic behavior.

The system is modeled by a continuous-time Markov chain (CTMC), shown in Figure 4.1, [Tri82] [SS82]. Since our analysis will focus on gracefully degradable systems, we will not consider system repair and therefore the CTMC is acyclic. The following parameters are used:

- $P_i(t)$ is the occupation probability of *state* i, i being the number of failed processors, $i = 0, \ldots, D-1, F$; where F is the state of total system failure.

- D is the number of allowable degradation states, expressed as a function of N.
- c_i is the a state-dependent coverage factor, which is the probability of successful recovery from a single failure in state i.

The rates of state transitions, λ_i and μ_i, can be expressed as a function of the single processor failure rate λ, as follows:

$$\lambda_i = c_i(N-i)\lambda$$

$$\mu_i = (1 - c_i)(N-i)\lambda$$

In the analysis which follows we will assume, for simplicity, a constant (*i.e.,* state independent) coverage factor.

$$c_i = c \qquad i = 0, \ldots, D-1$$

This assumption is somehow unrealistic since the probability of partitioning the network increases with increasing i and might reach values comparable to any assumed value of c (see Chapter 3).

Let $P_i(t)$ be the state occupancy probability of state i; in other words it is the probability of the system having exactly i failures. The steady-state solution of this Markov model is described by the following differential equations:

$$\begin{aligned}
\frac{dP_0(t)}{dt} &= -(\lambda_0 + \mu_0)P_0(t) = N\lambda P_0(t) \\
\frac{dP_i(t)}{dt} &= -(\lambda_i + \mu_i)P_i(t) + c\lambda_{i-1}P_{i-1}(t) \\
&= -(N-i)\lambda P_i(t) + c(N-i+1)\lambda P_{i-1}(t) \qquad i = 1, \ldots, D-1 \\
\frac{dP_F(t)}{dt} &= \lambda P_D(t) + \lambda(1-c)\sum_{j=0}^{D-1}(N-j)P_j(t)
\end{aligned}$$

Subject to the following constraints:

$$\begin{aligned}
P_0(0) &= 1 \\
P_i(0) &= 0 \qquad i = 1, \ldots, D-1
\end{aligned}$$

The state probabilities can be derived as:

$$\begin{aligned}
P_0(t) &= e^{-N\lambda t} \\
P_i(t) &= c^i \begin{pmatrix} N \\ N-i \end{pmatrix} (e^{-\lambda t})^{(N-i)}(1 - e^{-\lambda t})^i \qquad i = 1, \ldots, D-1
\end{aligned}$$

The reliability $R(t)$ is simply the probability of being in any one of the states $i = 0, \ldots, D-1$.

$$R(t) = \sum_{i=0}^{D-1} P_i(t) \tag{4.1.11}$$

The *mean time to failure* ($MTTF$) which is the expected time to first failure, is:

$$MTTF = \int_0^\infty R(t)dt \tag{4.1.12}$$

The Mission Time, MT, is defined for a given minimum reliability R_{min} as:

$$R(MT) = R_{min} \tag{4.1.13}$$

We define $F_P(t)$ to be the expected number of failed processors at time t:

$$F_P(t) = \sum_{i=0}^{D-1} iP_i(t) \tag{4.1.14}$$

Unless otherwise noted, in the rest of this discussion, we will assume a fully degradable system. This means that the system allows graceful degradation for up to $N-1$ failures, in other words, $D = N-1$. The unit-time will be taken as $1/\lambda = MTTF_1$ (i.e, the MTTF of a single processor) and a value of $R_{min} = 0.99$.

4.2 Time-Based Reliability Analysis

In this section, we present a reliability analysis of large-scale degradable systems based on two time measures: (1) the Mean-Time-To-Failure ($MTTF$) and (2) the Mission-Time (MT). The Mission Time measure is primarily intended to evaluate the reliability of mission-oriented applications such as non-repairable, on-board systems. It is used in this analysis as a measure of the time interval where $R(t) \geq R_{min}$.

In both cases we will use the results to evaluate the interval T_{gss} assuming it can be expressed as a function or a fraction of the $MTTF$ or the MT respectively.

4.2.1 MTTF-based evaluation

The expression for $MTTF$ can be obtained from Equations 4.1.12 and 4.1.11 as:

$$MTTF = \int_0^\infty \sum_{i=0}^{D-1} P_i(t)dt = \frac{1}{\lambda c} \sum_{i=1}^{D-1} \frac{c^i}{i} \tag{4.2.15}$$

The expression for $MTTF$ indicates that increases in N have diminishing effects on the value of the expected time to first failure. In fact, an increase from N to $N+1$ processors results in a minimal increase in $MTTF$:

$$MTTF(N+1) - MTTF(N) = \frac{c^{N+1}}{N+1}$$

since $c < 1$, the increase becomes insignificant for large values of N.

The values of $MTTF$ are plotted in Figure 4.2 as a function of N for different values of c. The series in Equation 4.2.15 is not convergent but has a logarithmic behavior. Therefore there is no asymptotic limit to $MTTF$. However, for all practical purposes, the mean time to failure can be considered constant for sufficiently large N given a value of the coverage factor.
Let N_k be the value of N at the knee of the curve in Figure 4.2, i.e

$$MTTF(c, N) \approx MTTF(c, N_k) \qquad for \quad N > N_k$$

An analysis of the values in Figure 4.2 shows the following:

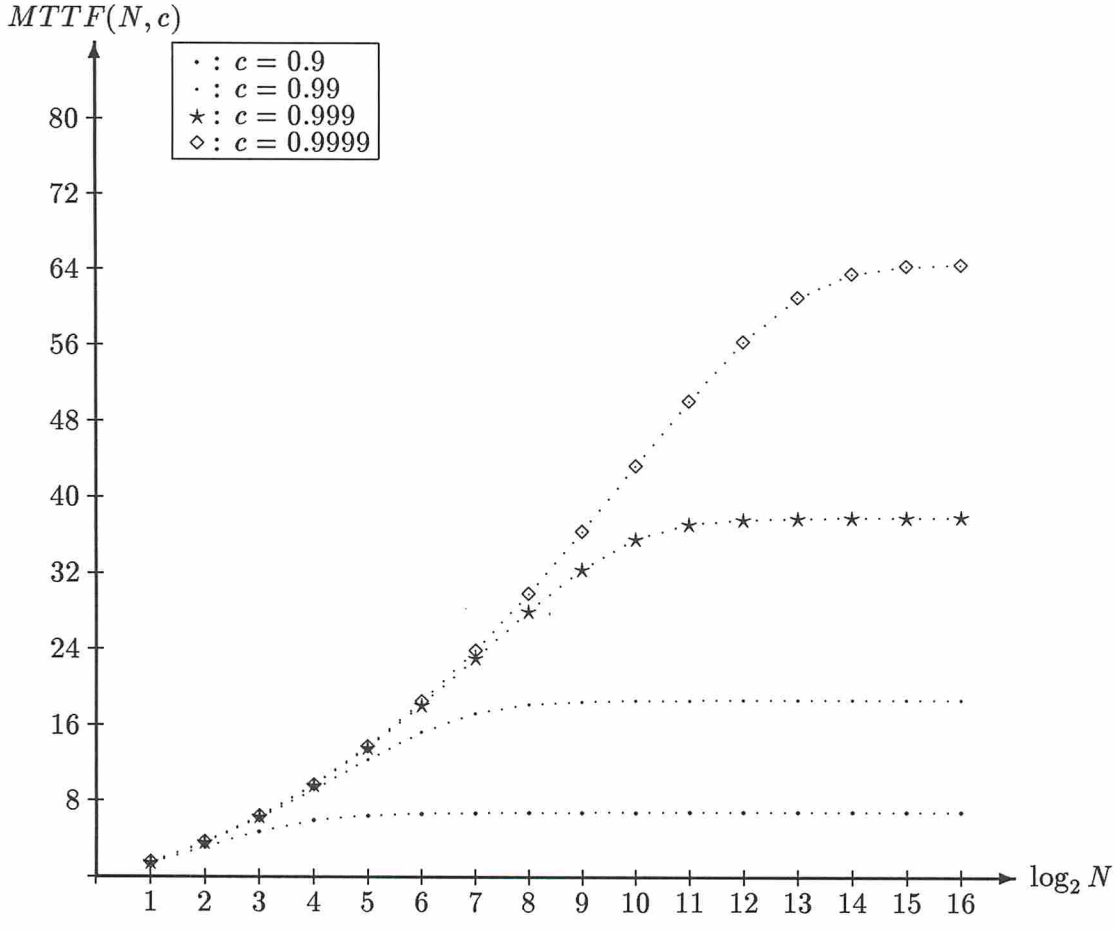


Figure 4.2: $MTTF$ as function of N and c

- The effects of a larger N on the $MTTF(N, c)$ are significant compared to the MTTF of a single processor. For example, for $c = 0.99$ and $N = 128$ the expected time to first failure is 16 times that of a single processor, or $MTTF(128, 0.99) = 16\ MTTF(1)$.

- For smaller values of N the coverage c has little effect on the values of the MTTF. For example, for $N = 32$, the expected time to first failure $MTTF(32, c) \approx 13$ for all values of $c \geq 0.99$.

- As the probability of failed recovery, $(1 - c)$, is decreased by a factor of 10 the corresponding increase in $MTTF$ is ≈ 2. For example , for $N > 128$ and an increase in c from 0.9 to 0.99 results in an increase in MTTF from 6.8 to 18.6 (factor of 2.7). This means that substantially large improvement on the reliability of the recovery procedure do not significantly affect the expected time to failure.

From the above analysis, we can conclude that:

1. For smaller systems, the probability of successful recovery has very little effect on the mean time to first failure.

2. For larger systems, the mean time to first failure is a constant function of the coverage factor and is independent of the number of processors.

An $MTTF$ based evaluation of T_{gss} implies that for a given value of c and $N > N_k$, the interval can be kept constant independently of the number of processors. However, as N increases, the amount of *computational work* performed during that interval increases and the computation progresses faster.

A drawback of an $MTTF$ based evaluation is that it cannot take into account the overall system reliability. Of particular is the system reliability at the time when the global state saving procedure is performed. In fact, if $R(T_{gss})$ is not high enough the system might have crashed at $T < T_{gss}$ or the states to be saved might be corrupted which defeats the whole purpose of global state saving.

4.2.2 MT-based evaluation

The *Mission-Time* is defined as the time interval where $R(T) \geq R_{min}$, therefore:

$$R(MT) = R_{min}$$

The values of MT, as obtained from Equation 4.1.13, are plotted in Figures 4.3 and 4.4 as a function of N for different values of c. The same results are reported numerically in Table 4.1.

These curves show that for a given value of c, there exists a value of N at which the MT is maximal. We denote this value by N_p.

$$MT(c, N_p) \geq MT(c, N) \quad \forall N$$

It is clear from these curves that for smaller values of N ($N < N_p$) the inherent redundancy of the system provides a higher mission time. As N increases ($N > N_p$) the higher failure rate dominates and reduces the mission time. Furthermore, as c is increased, the value of N_p also increases (see Figure 4.4).

From the analysis of these results, we can deduce the following:

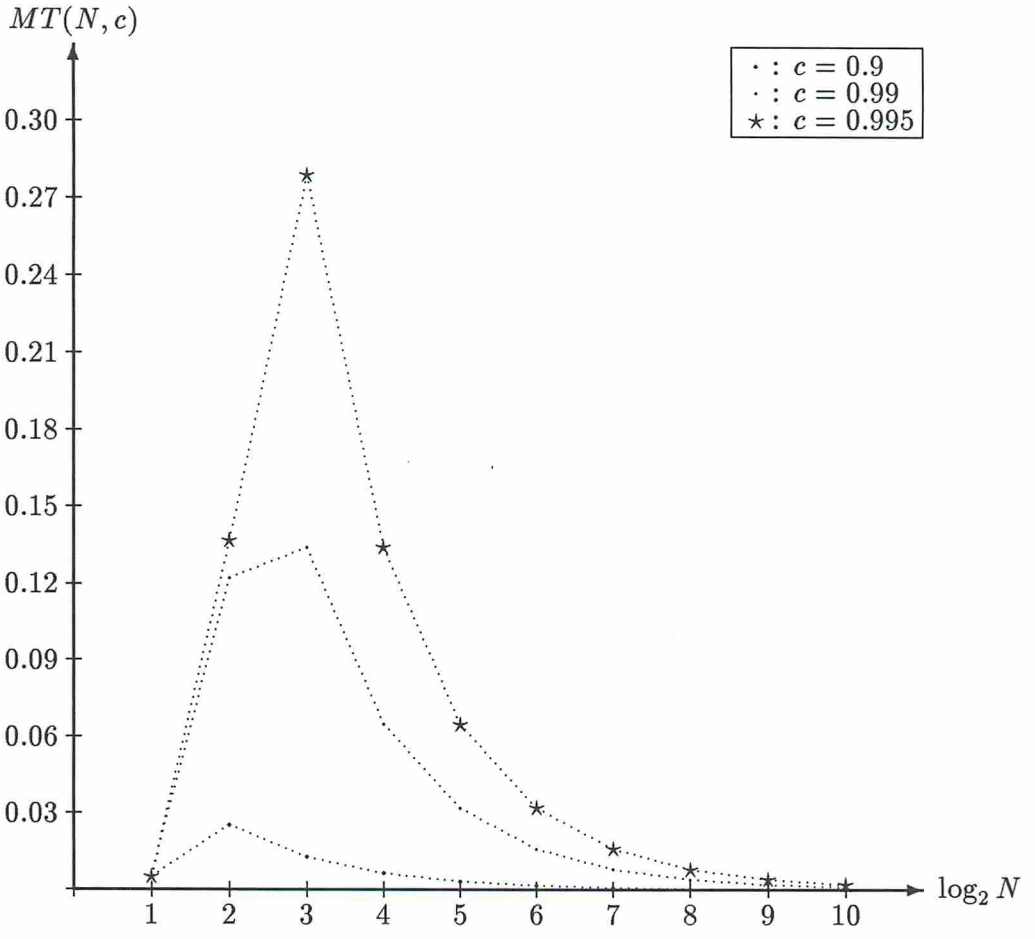


Figure 4.3: Mission-Time as function of N ($D = N - 1$, $c < 0.995$)

N	c			
	0.9999	0.999	0.99	0.9
2	0.11	0.105	0.096	**0.044**
4	0.38	0.37	**0.232**	0.025
8	0.82	0.75	0.134	0.013
16	1.37	**0.93**	0.065	0.0063
32	1.95	0.38	0.032	0.0032
64	**2.51**	0.17	0.016	0.0016
128	1.54	0.08	0.008	0.0008
256	0.5	0.04	0.004	0.0004
512	0.22	0.02	0.002	0.0002
1024	0.11	0.01	0.001	0.0001

Table 4.1: Mission-Time $D = N - 1$ and $R_{min} = 0.99$

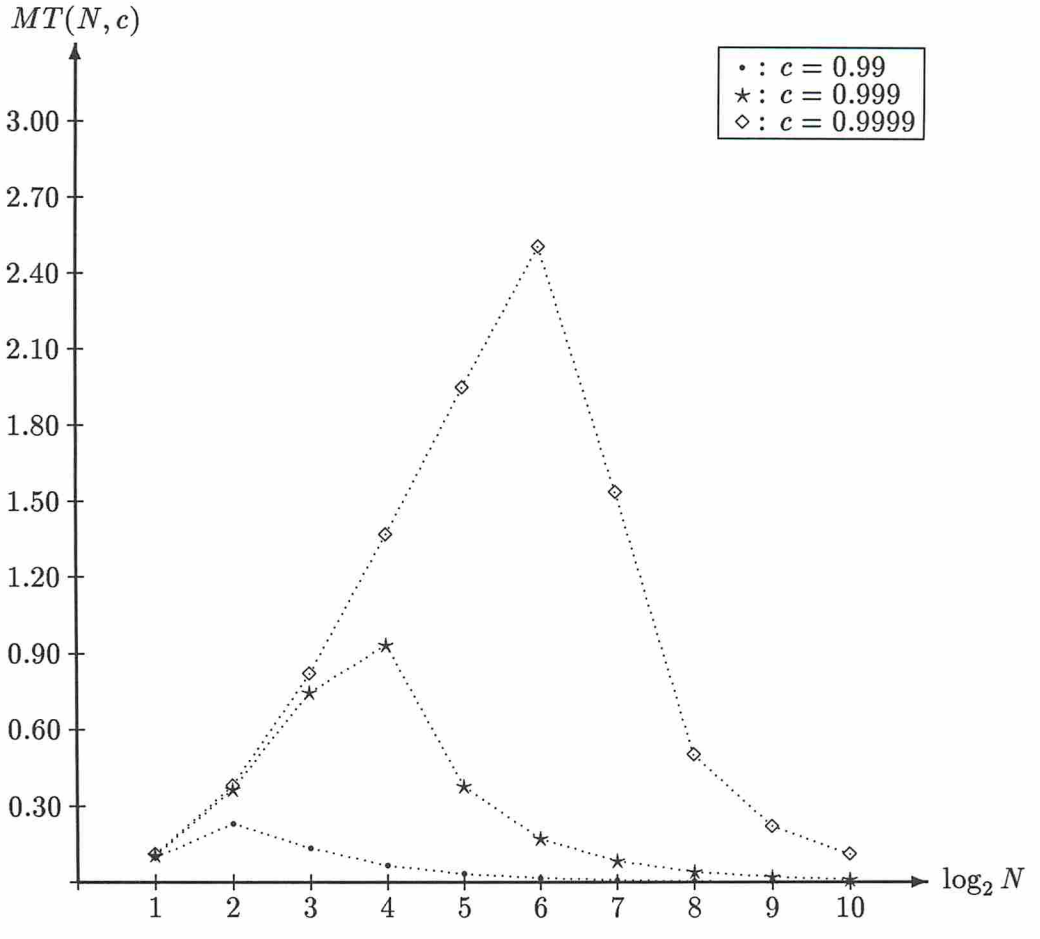


Figure 4.4: Mission-Time as function of N ($D = N - 1$, $c > 0.99$)

- The plot of the mission-time as a function of the number of processors, N, for a gracefully degradable system shows that the mission-time reaches a peak value at $N = N_p$.
- The peak value of the mission-time is significantly larger than that of a single processor. For example: for $R_{min} = 0.99$ the mission -time of a single processor, $MT(1) = 0.01$, therefore $MT(N_p, 0.999) \geq 93MT(1)$.
- As the number of processors is increased beyond N_p, i.e $N > N_p$, we observe a decrease in the mission-time. This decrease can be observed to be inversely proportional to N. That is $MT(2N, c) = 0.5MT(N, c)$.
- While the peak value of the mission-time of a multicomputer system can be significantly larger than that of a single processor, the reverse becomes true for very large values of N. For example, from Table 4.1 we can observe that $MT(1) = 10MT(1024, 0.99)$.
- From Table 4.1 we can observe that, for $N > N_p$, a 10 fold decrease in $(1-c)$ (the probability of failed recovery) results in a 10 fold increase in the MT for the same number of processors. For example, $MT(128, 0.999) = 10MT(128, 0.99)$. In other words, the mission-time is inversely proportional to $(1 - c)$.

From these numerical results we can deduce the following proportionality expression:

$$MT \propto \frac{1}{(1-c)N} \tag{4.2.16}$$

While this expression is empirically based on the numerical results for $D = (N-1)$ and $R_{min} = 0.99$, it will be analytically justified in section 4.3.3.

An MT based evaluation of T_{gss} implies that an increase in N is not always beneficial to the total computation time. In fact, for $N > N_p$ it could be detrimental since the T_{gss} interval is reduced as N increases while the time overhead of GSS procedure itself increases too. However, since $R(T_{gss}) \geq R_{min}$ this approach, offers the advantage of guaranteed high reliability when the GSS procedure is performed and therefore the correctness of the saved states.

This analysis, however, does not indicate whether the decrease in T_{gss}, for $N > N_p$, is offset by the increase in execution speed, and therefore of computational work. This problem is addressed in the next section where we analyze the effects of c on the measure of *computational work* and *reliable computational work.*

4.2.3 Expected Number of Failures

In this section we propose an analytical evaluation of the expected number of failures in the time interval $[0, MT]$. This analysis will allow us to derive the value of N_p where the MT is maximal as observed in section 4.3.2.

Because of the cumulative effects of the probability of successive recovery, after the i^{th} failure, the reliability of the system is constrained by:

$$R(t) \leq c^i$$

Let K' be defined such that:

$$c^{K'} = R_{min}$$

Therefore, an integer value of K', K, can be derived as:

$$K = \left\lfloor \frac{\log R_{min}}{\log c} \right\rfloor \tag{4.2.17}$$

Given the definition of MT, K' is the expected number of failures in the interval $[0, MT]$ constrained by the condition that $D \geq K'$. In other words, if the number of processors is large enough, K failures are sufficient to reach $R(t) = R_{min}$. Since K, as defined in equation 4.2.17, can take only integer values, it is an approximation of the expected number of failures K'.

From this analysis, we can deduce that when the number of allowed degradation states is sufficiently large (i.e, $D \gg K$), the necessary condition to reach the minimum reliability level ($R(t) = R_{min}$) and therefore the mission-time is that K processors fail. Since the rate of failures is proportional to the number of processors, the time interval $[0, MT]$ is inversely proportional to N. On the other hand, for ϵ very small and $x = 1 - \epsilon$, we can use the following approximation for $\log x$:

$$\log x \approx 1 - x$$

We have therefore demonstrated that for $D \gg K$:

$$MT \propto \frac{1}{(1-c)N}$$

4.3 Computation-Based Analysis

In this section we present an evaluation of performance and reliability of large-scale degradable systems based on the notion of *computational work*. There is no formally defined unit of computational work. In this analysis we will use *processor-hours* as units of computational work. Other measuring units could be machine instructions. Any computational task is characterized by a certain amount of computational work, measured in processor-hours. When this task is executed over several processors, the execution time is reduced, but the amount of processor-hours required for that computation is kept constant if the speed-up is linear. For non-linear speed-ups the amount of required processor-hours increases due to added overhead.

Let T_n be the execution time of a given computation over n processors. S_n is the attainable speed-up defined by:

$$S_n = \frac{T_1}{T_n}$$

S_n is equivalent to the number of *effective processors* (i.e the number of virtual processors fully utilized by the given computation). We define $CW(N,t)$ as the amount of *effective computational work* a system will deliver for a given computational speed-up.

$$CW(N,t) = \int_{\tau=0}^{t} \sum_{i=0}^{D-1} S_{(N-i)} P_i(\tau) d\tau \tag{4.3.18}$$

Based on the model described in section 4.2, we define $PH(N,t)$ as the amount of processor-hours a system, with initially N processors, can deliver up to time t, as:

$$PH(N,t) = \int_{\tau=0}^{t} \sum_{i=0}^{D-1} E_{(N-i)}(N-i) P_i(\tau) d\tau \tag{4.3.19}$$

For a computation that exhibits linear speed-up (i.e. $S_n = n$) we have: $CW(N,t) = PH(N,t)$. For the sake of simplicity, in the rest of this discussion, we will assume a best case of linear

speed-up and therefore $E_N = 1$. Note that $PH(N,\infty)$ is the *mean computation before failure* (MCBF) and $CW(N,t)$ is the integral of the *computational availability*, $a_C(t)$, as defined in [Bea78]. Both $PH(N,t)$ and $CW(N,t)$ are *expected* values of the processor-hours and computational work measures.

In section 4.4.1 we show that, in a gracefully degradable system, the amount of computational work, $PH(N,t)$, is upper-bounded. The upper-bound is independent of N, i.e

$$PH(N,t) < PH_{max} \qquad \forall N, \forall t$$

Using $PH(N,t)$, we define, in section 4.4.4, the measure of *Reliable Computational Work*, $RCW(N)$, and show how it can be used to evaluate T_{gss}.

4.3.1 Upper Bound on PH

In this section, we prove that the amount of computational work a purely degradable system can deliver is upper bounded and that the upper bound is independent of the initial number of processors. We derive this upper bound using (1) a discrete analysis and (2) a continuous-time Markov model.

Theorem 4.1 *$\forall N$ and $c < 1$ $\exists$ PH_{max} such that*
$PH(N,t) < PH_{max}$, $\forall t$.

4.3.1.1 Proof 1: Continuous-time Markov model

Using the binomial theorem, Equation 4.1.11 can be rewritten as:

$$P_i(t) = c^i \binom{N}{i} \sum_{k=0}^{i} \binom{i}{k} (-1)^{(i-k)} (e^{-\lambda t})^{(N-k)} \tag{4.3.20}$$

Therefore, Equation 4.3.19 can be rewritten as:

$$PH(N,t) = \int_0^t \sum_{i=0}^{D-1} (N-i) c^i \binom{N}{i} \sum_{k=0}^{i} \binom{i}{k} (-1)^{(i-k)} (e^{-\lambda \tau})^{(N-k)} d\tau \tag{4.3.21}$$

Integrating over τ, and taking the limit as $t \longrightarrow \infty$, we obtain:

$$PH(N,\infty) = \frac{1}{\lambda} \sum_{i=0}^{D-1} (N-i) c^i \binom{N}{i} \sum_{k=0}^{i} \binom{i}{k} (-1)^{(i-k)} \frac{1}{(N-k)} \tag{4.3.22}$$

The second summation in Equation 4.3.22 can be transformed using binomial identities into:

$$\sum_{k=0}^{i} \binom{i}{i-k} (-1)^{(i-k)} \frac{1}{(N+i)+(i-k)} = \frac{1}{(N-i)\binom{N}{i}}$$

Equation 4.3.22 is reduced to:

$$PH(N,\infty) = \frac{1}{\lambda} \sum_{i=0}^{D-1} c^i = \frac{1-c^D}{\lambda(1-c)} \tag{4.3.23}$$

Therefore:

$$PH_{max} = \frac{1}{\lambda} \frac{1}{1-c} \tag{4.3.24}$$

and $PH(N,\infty) < PH_{max} \quad \forall N$. □

4.3.1.2 Proof 2: Discrete Analysis

The amount of processor-hours, PH, can also be expressed as a function of the number of failures i. $PH(i)$ is therefore the amount of processor-hours at the i^{th} failure.

$$PH(1) = \frac{N}{N\lambda} = \frac{1}{\lambda}$$

$$PH(i) = cPH(i-1) + \frac{N-i-1}{(N-i-1)\lambda} = cPH(i-1) + \frac{1}{\lambda} \tag{4.3.25}$$

From Equation 4.3.25 we can rewrite $PH(i)$ as:

$$PH(i) = \frac{1}{\lambda}(1 + c + c^2 + ... + c^{(i-1)}) = \frac{1}{\lambda}\frac{1-c^i}{1-c} \tag{4.3.26}$$

Therefore:

$$PH_{max} = \lim_{i \to \infty} PH(i) = \frac{1}{\lambda}\frac{1}{(1-c)} \tag{4.3.27}$$

Therefore PH_{max} is a constant upper bound on $PH(i)$ as $i \longrightarrow \infty$. □

The conclusion from Theorem 1 is that no matter how large the initial number of processors is, there is an upper bound on the amount of processor-hours that are obtainable when $c < 1$. This upper bound is determined by c only and is reached asymptotically.

PH_{max} is therefore the upper limit on the *mean computation before failure* (MCBF). Comparing this result to that in section 4.3.1 shows that while the expected time to failure of a degradable system increases logarithmically with N, the expected computational work performed in that interval is upper bounded for all N. Therefor increasing the system size does not increase the amount of expected computational work the system can deliver before total failure.

4.3.2 Reliable Processor-Hours

The measure of *reliable processor-hours*, RPH, is defined as the amount of processor-hours available while the reliability is maintained above a given minimum, i.e:

$$RPH(N,c) = PH(N,MT) = \int_{t=0}^{MT} \sum_{i=0}^{D-1} (N-i)P_i(t)dt \tag{4.3.28}$$

The results of evaluating the RPH, according to Equation 4.3.28, are presented in Table 4.2, for $D = N/2$, and Table 4.3 for $D = N-1$. The values of c have been chosen in the range $[1, 0.99]$. The value of R_{min} has been set to 0.99. The values of $RPH(N,c)$ are expressed in $processor - hours$ where the unit time is taken as $1/\lambda$.

Two observations can be made:

1. for a given value of c, there is a value of N beyond which an increase in N will not increase the amount of reliable processor-hours. We denote this value by $N_{ph}(c)$. For example, $N_{ph}(0.999) = 64$. We can formally define $N_{ph}(c)$ as:

$$\forall c < 1, \quad N \geq N_{ph}(c) \Longrightarrow RPH(N,c) = RPH_{max}(c)$$

2. the values of N_{ph} and RPH_{max} increase with increasing values of c. For example, in Table 1, $N_{ph}(0.99) = 16$ and $RPH_{max} = 1.0$, $N_{ph}(0.999) = 64$ and $RPH_{max} = 10.0$ and $N_{ph}(0.9999) = 512$ with $RPH_{max} = 100$.

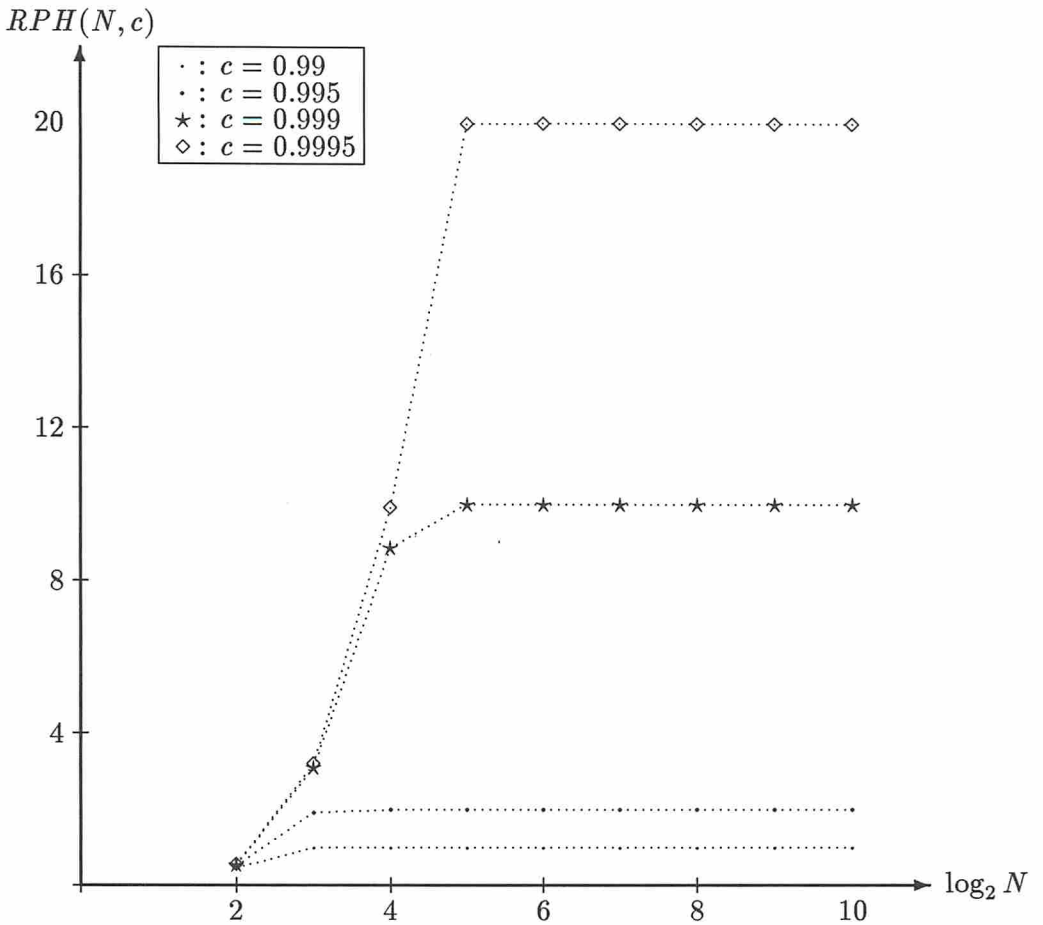


Figure 4.5: RPH as function of N and c

N	c					
	1.0	0.9999	0.9995	0.999	0.995	0.99
4	0.168	0.168	0.168	0.168	0.164	0.156
8	0.966	0.963	0.955	0.94	0.84	0.70
16	3.38	3.35	3.29	3.2	1.95	**1.00**
32	9.2	9.15	8.8	8.0	**2.00**	1.00
64	22.4	22.1	19.0	**10.0**	2.00	1.00
128	50.3	49.0	**20.0**	10.0	2.00	1.00
256	106.7	98.0	20.0	10.0	2.00	1.00
512	220	**100**	20.0	10.0	2.00	1.00
1024	455	100	20.0	10.0	2.00	1.00

Table 4.2: Reliable Processor-hours for $D = N/2$ and $R_{min} = 0.99$

N	c					
	1.0	0.9999	0.9995	0.999	0.995	0.99
4	0.563	0.562	0.558	0.553	0.511	0.459
8	3.28	3.26	3.19	3.1	1.93	**1.0**
16	10.42	10.33	9.89	8.86	**2.0**	1.0
32	25.93	25.62	19.98	**10.0**	2.0	1.0
64	57.67	56.64	**20.0**	10.0	2.0	1.0
128	122.77	**100.3**	20.0	10.0	2.0	1.0
256	250.7	100.0	20.0	10.0	2.0	1.0
512	508.0	100.0	20.0	10.0	2.0	1.0
1024	1019.0	100.0	20.0	10.0	2.0	1.0

Table 4.3: Reliable Processor-hours for $D = (N - 1)$ and $R_{min} = 0.99$

	c				
	0.9999	0.9995	0.999	0.995	0.99
K	100	20	10	5	1
RPH'	99.5	19.9	9.95	1.995	1
RPH_{max}	100	20	10	2	1
Ph_{max}	10^4	2000	1000	200	100

Table 4.4: Comparison of RPH' and RPH_{max} for $R_{min} = 0.99$

Theorem 1 states that the expected amount of processor-hours in the interval $[0, \infty]$ is upper bounded by PH_{max}. These results show that the expected amount of processor-hours in the interval $[0, MT]$ is also upper bounded, the upper bound being RPH_{max}.

RPH_{max} is therefore the maximum expected amount of computational work the system can deliver subject to the constraint of $R(t) \geq R_{min}$. In the next section we present an analytical derivation of $RPH_{max}(c)$.

The maximum value of $RPH(N, c)$, $RPH_{max}(c)$, can be derived analytically by using the expression for the expected number of failures in the interval $[0, MT]$ as defined in section 4.3.3.

$$RPH_{max} \approx RPH' = \frac{1 - c^K}{1 - c} \tag{4.3.29}$$

Note that Equation 4.3.29 is an approximation of RPH_{max} because K can take only integer values while MT in Equation 4.3.28 has real values.

Table 4.3.2 shows the values of K, RPH', RPH_{max} and Ph_{max} for various values of c. From the values of RPH_{max} and RPH', we can observe that the approximation of Equation 4.3.29 is accurate within 5%. Therefore, Equations 4.3.29 and 4.2.17 provide a simple method for deriving RPH_{max}, given the values of c and R_{min}.

4.3.2.1 Discussion of Results

The results of section 4.2 show that there is no increase in reliable computational work when N is increased above N_{ph} for a given value of c. This confirms the results obtained in the MT based evaluation. Although the data reported in Figures 4.3 and 4.4 and in Tables 4.2 and 4.3 covers

only a few values of N, it can be noted that the values of N_p, where MT is maximal, and those of N_{ph}, where $RPH(N, c)$ is constant, are very closely related.

It appears, therefore, that for a given value of the coverage c there exists an optimal value of N, N_{opt}, that would maximize the mission-time MT, and therefore the interval T_{gss}, while preserving a high reliability. From the numerical data presented in this chapter we cannot derive an *exact* value for N_{opt}, however, the interval can easily be narrowed down. For example, for $c = 0.999$ and $D = N - 1$ we have:

$$16 \leq N_{opt} < 32$$

4.3.3 Reliable Computational Work

RPH evaluates the amount of reliable processor-hours potentially available from the system. The fraction of RPH that is actually used by a computation depends on the speed-up S_n of the computation. The speed-up of a computation is equivalent to the number of *effective processors* or the number of *virtual fully-utilized* processors. It is defined as:

$$S_n = \frac{T_1}{T_n}$$

where T_i is the execution time over i processors. When $S_n = n$ the computation is said to exhibit linear speed-up. This implies that the communication and synchronization overhead in that computation is negligible compared to the execution time. When $S_n < n$ the speed-up is said to be sub-linear.

Similarly to RPH we define RCW as:

$$RCW(N, c) = CW(N, MT)$$

RCW is therefore the amount of *effective reliable computational work* a system can deliver with respect to a given computation while $R(t) \geq R_{min}$. Therefore $RPH = RCW$ for $S_n = n$.

In evaluating RCW, we will take as example a sub-linear speed-up case where:

$$S_n = \frac{n}{\log n}$$

The results, plotted in Figure 4.6, show that:

$$\exists N_r \ \ such\ that \ \ RCW(N_r) \geq RCW(N) \quad \forall N$$

In other words, there exists a value of N denoted by N_r at which the value of RCW is maximal.

Figure 4.7 shows the plot of both RPH and RCW versus N for $c = 0.995$ and $S_n = \frac{n}{\log n}$. It can be noted that $N_r = N_{ph}$ as reported in Table 4.3 (for $D = N - 1$). This effect is predictable: since RPH is constant for $N > N_{ph}$ and a linear speed-up, for a sub-linear speed-up RCW must be a decreasing function of N.

This implies that as the system size is increased over N_{ph}, the probability of a computation not completing reliably decreases if the speed-up of the computation is sub-linear.

This result has implications on the scalability of graceful degradation. For a large-scale gracefully degradable system to be scalable, any increase in the system size should be matched by an increase in the quality of the recovery scheme, i.e the coverage factor, in order to maintain the same performability level.

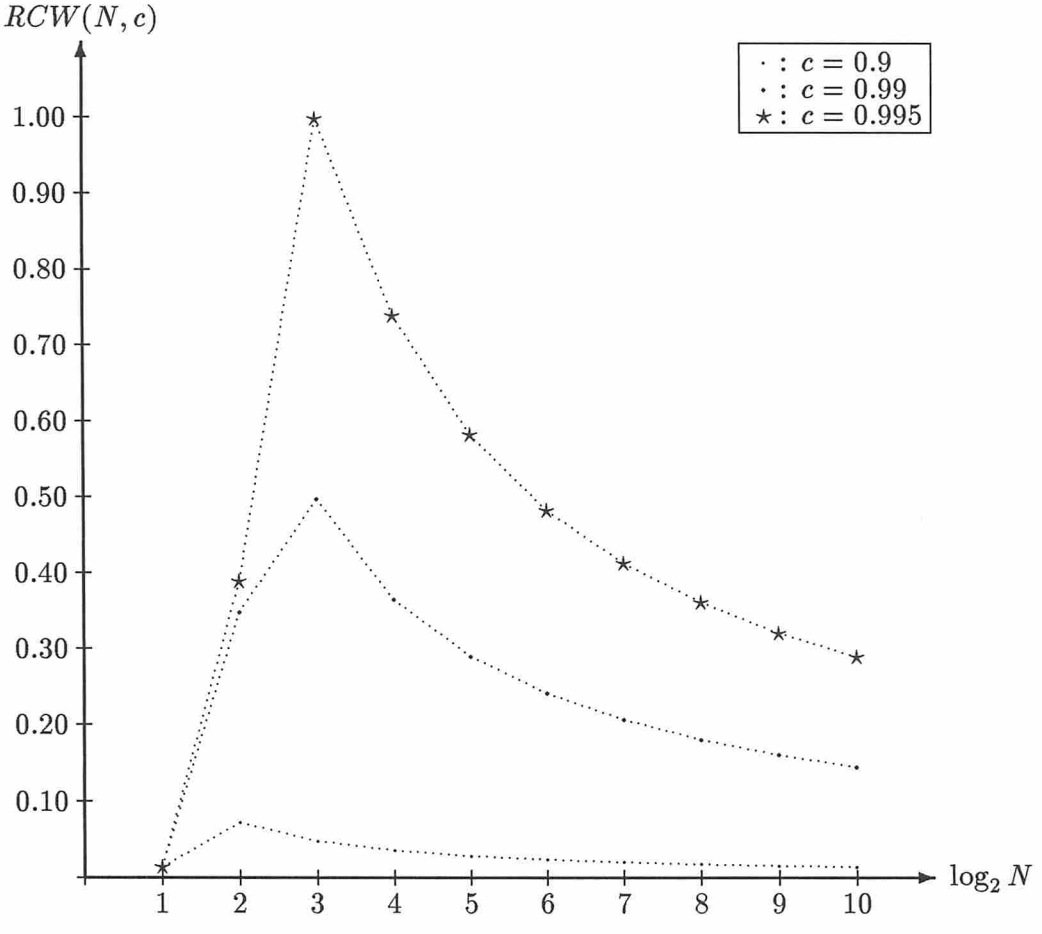


Figure 4.6: RCW as function of N and c for $S_n = \frac{n}{\log n}$

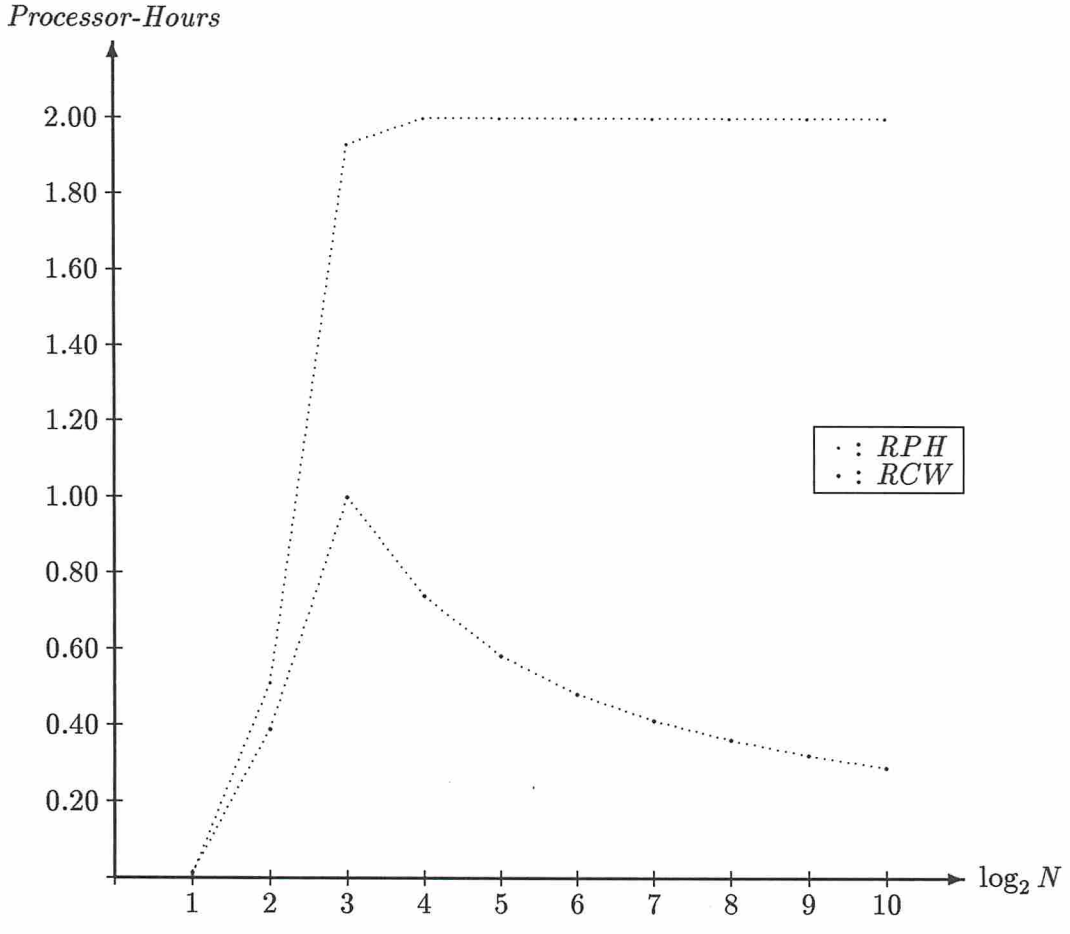


Figure 4.7: RCW and RPH as function of N, $c = 0.995$ and $S_n = \frac{n}{\log n}$

4.4 Discussion of the Results

The results of the analysis performed in this chapter point out to the fact that gracefully degradable large-scale systems do not scale-up. If a minimum reliability level is to be maintained throughout, then there exists a system size that would provide the optimal mission time and amount of reliable computational work. For any larger system there would be a decrease in either performance (as expressed in computational work) or reliability (as represented by the minimum level).

Throughout this analysis, one parameter has been maintained constant and used a measure of unit time, that is the mean-time-to-failure of the single processor ($MTTF_1$). We have therefore assumed that the systems were built using the same set of basic building blocks. A system architect, however, has a wide range of choices for his basic building blocks. The technologies range from high-speed high-power ECL technology to low-speed low-power CMOS technologies. In addition to the speed and power consumption factors, technologies can also be classified by their age. Old technologies have time-proven characteristics and well established design rules and methodologies. Their parameters have typically very narrow tolerance levels and they generally have a very low failure rate. New technologies, on the other hand, are more prone to either design errors or higher component failure rates.

Therefore, the choice of a technology by a system designer not only determines the allowable switching speed, and thereby the potential computing power of the system, but also the expected rate of failure [1]. By determining the failure rate, the age of a technology determines the assumed unit-time $MTTF_1$. While new technologies can offer switching speeds several orders of magnitude larger than the older technologies, these can often be several orders of magnitude more reliable. It is conceivable, therefore, that a system built with a time-proven but slow-switching technology might actually outperform a system built with faster but less reliable technology.

[1] Another major relevant factor to this choice is the cost of design and components associated with a given technology. This factor, however, is not of immediate relevance to the analysis proposed in this chapter.

Chapter 5

ALGORITHM FOR DISTRIBUTED FAULT-TOLERANCE

"We have another bad ... unit. My fault predictor indicates failure within twenty-four hours".

"I don't understand it, Hal. Two *units can't blow in a couple of days."*

"It does seem strange, Dave. But I assure you there is an impending failure."

Arthur C. Clarke
2001: A Space Odyssey

In Chapter 4, we have shown that the performance and reliability of gracefully degradable systems do not scale up as the number of processing elements is increased, unless the quality of the protection scheme, i.e. the coverage factor, is increased accordingly. A highly fault-tolerant protection scheme for large-scale systems should be immune to the effects of node disconnections. In fact, the results in Chapter 3 show that the probability of disconnection can become a significant threat in large systems with constant connectivity.

In this chapter we propose a methodology for implementing distributed checkpointing and an algorithm based on that methodology, that implements distributed fault-tolerance. The methodology for distributed checkpointing is based on the semantic properties of functional program execution.

We review the concepts and properties of functional program execution and present the methodology that allows program checkpointing to be implemented in a distributed fashion at run-time. A simple iterative distributed algorithm for fault-tolerance based on functional program execution is then described. The performance of this algorithm is then evaluated and techniques for implementing system level diagnosis, based on this algorithm, are proposed.

5.1 Distributed Checkpointing in Functional Execution

In this section we describe a methodology that allows the distributed checkpointing of programs at run-time. This methodology is based on the Church-Rosser property of functional program execution. In section 5.1.1 we review some of the basic concepts and properties of functional languages and functional program execution. The proposed methodology is described in section 5.1.2.

5.1.1 Functional languages and execution

Functional languages have been proposed as an alternative to imperative languages mainly because of their ability to exactly model mathematical expressions [Bac78]. A functional language consists of a set of primitive functions and primitive data objects (atoms). A set of operators allow operations on functions such as composition, inversion, and the construction of complex data objects such as lists. A program in a functional language consists of a set of function applications on some data objects. Unlike imperative languages, the execution of functional languages is *side-effect free.*

The various properties of functional languages have been widely discussed in the literature. These include the possibility of algebraic program verification and program derivation from specifications. Probably the most popular and most interesting property as far as program execution is concerned is the Church-Rosser property which is an immediate consequence of the property of side-effect free execution. This property states that the order of function application in the execution of a functional program does not have any effect on the outcome of the program.

While the Church-Rosser property was derived in the context of functional languages, it is not a characteristic of the language itself as much as a that of the *execution model.* In fact, this property holds as long as side-effect free execution is guaranteed. In a *functional execution model,* a program is modeled as a set of side-effect free tasks. A task executes on a set of input data objects and produces a set of output data objects in a function like way. This property, therefore, holds irrespective of the actual programming language used as long as the functionality of the execution is preserved.

Based on the referential transparency property, we propose a model for the functional execution of distributed programs on multicomputer systems. In this model a program executes as a set of distributed communicating tasks. The task is the smallest execution entity; it executes as a process on a single processor. Data is communicated between tasks as messages. All incoming messages are received by a task before execution starts. All outgoing messages are sent upon completion of execution. Task execution is functional in that it has no effect on the program except through its outgoing messages. In the next section, we describe how such a model can support distributed checkpointing.

5.1.2 A Distributed Checkpointing Methodology

A reliable checkpointing mechanism is an essential element in any fault recovery scheme. In a uniprocessor, a consistent checkpoint is a snapshot of the state of the program at a given time. It can be achieved, conceptually, by saving the state of the memory and all relevant registers. In a distributed system, however, the problem of checkpointing is rendered more complex by the asynchronous nature of the execution and the communication delays among processors.

The methodology for distributed checkpointing presented in this section was originally proposed as the Token Resending scheme by Gaudiot *et al.* [GR85] for the reliable execution of data-flow programs.

The scheme is as follows: whenever a task completes execution, a copy of all the output messages that are produced is kept by the processor on which the task executes. At the same time, a special acknowledgement message (*ack*) is sent to all the processors from which messages have been received as input messages to that task. The reception of an *ack* message results in the deletion of the corresponding copy of the message. Figure 5.1 illustrates this scheme. Task C starts executing on processor P_3 when both tasks A and B have terminated and messages containing the data values x and y are received. Copies of the values of x and y are kept as x' and y' in processors P_1 and

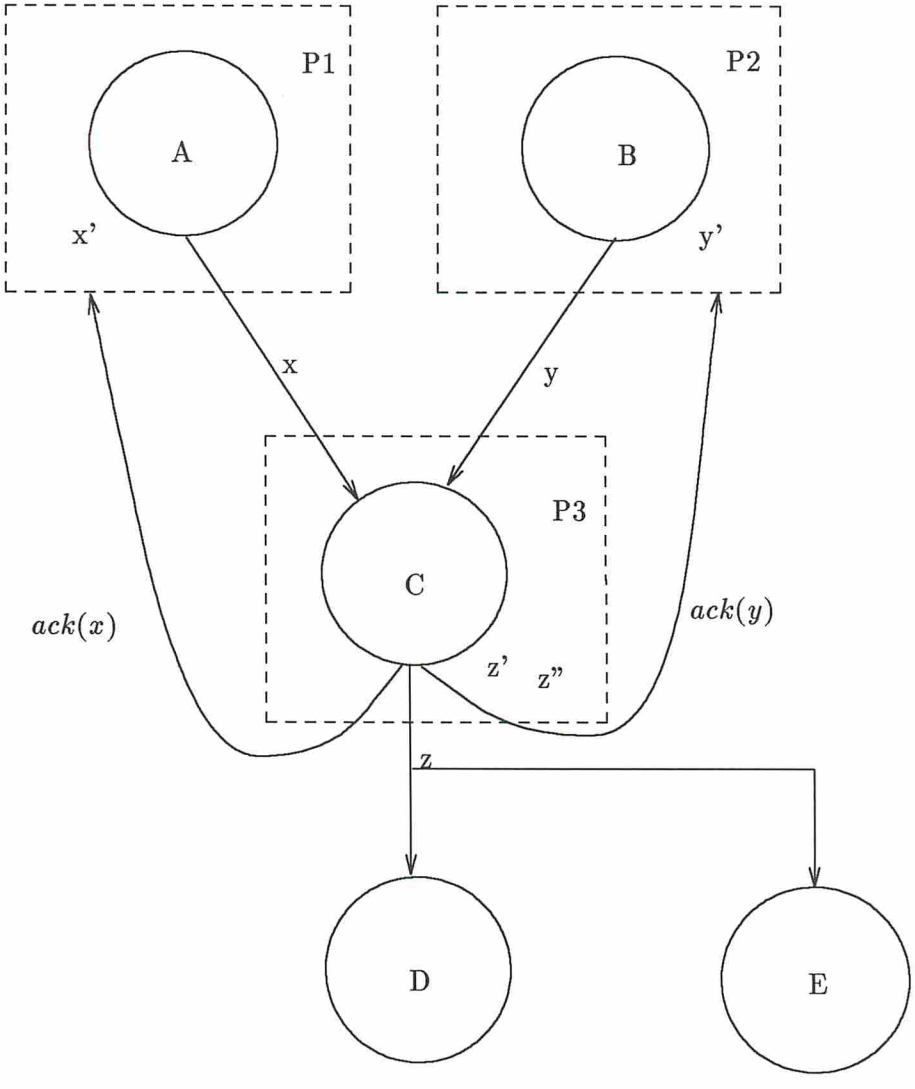


Figure 5.1: Example of distributed checkpointing

P_2 respectively. When task C terminates, the result value z is sent to tasks D and E. Two *ack* messages are sent to processors P_1 and P_2. That results in the deletion of the backup values x' and y'. If processor P_3 had failed before the completion of task C and the failure was detected and notified to processors P_1 and P_2, then task C could have been restarted on another processor with the same set of input data. Note that since the data value z is sent to two different destinations, two backup copies, z' and z'' are kept in P_3. A constraint imposed by this mechanism on task allocation is that two successive tasks should not be allocated to the same processor. Whenever, for efficiency considerations, two successive tasks must be allocated to a same processor, then they are are checkpointed as a single combined task.

Since any task that has not completed execution successfully can be restarted at any time from the backup copies of its input data set, the set of all backup data values constitutes the global state, or checkpoint, of the program. This checkpoint can be visualized as a wavefront advancing through the program execution graph. This mechanism allows for the checkpointing to be performed asynchronously at run-time along with the program execution. It does not, however, allow for the detection of failures in the system. The algorithm proposed in the next section exploits the properties of functional execution to implement fault detection and recovery.

5.2 An Iterative Algorithm

In this section we propose a distributed algorithm that is based on the checkpointing methodology described in section 5.1.2. This algorithm relies on both dual space redundancy and multiple time redundancy. the redundant execution serves the dual purpose of (1) improving the reliability of a task execution, and (2) detecting failures or transient faults.

Figure 5.2 is a description of the algorithm. Every task on pairs of processors and the results are compared. If a match occurs, the execution of the program proceeds by sending the output data to successor tasks. If the results do not match, the task is re-executed on another pair of processors. At every iteration where no match occurs, an *iteration-number* is incremented and passed along with the input data set. This number is used to determine the address of the next pair of processors in case of a failed match. In this fashion, a task is not executed twice on the same pair of processors. The *error-number* value is also incremented at each mismatch and is used to implement diagnosis algorithms that are described in section 4.

This algorithm has the following properties:

- Being an iterative algorithm, the outcome at each iteration is independent of the previous ones, therefore there is no reliance on a recursive history or results to determine a possible match as in the RAFT algorithm [Agr85].
- By relying on both time and space redundancy, the algorithm allows the detection of both transient and permanent faults.
- The algorithm has a relatively high cost of hardware overhead. The number of processors used after iteration i is $2i$ whereas it is $i+1$ in the recursive algorithm. However, this cost is still substantially smaller than that of a TMR scheme where a voter device would be needed.

The fact that the iterative algorithm does not rely on a recursive data structure of previous outcomes makes it suitable for distributed execution. Multicomputer systems are often structured as regular topologies such as a mesh, binary n-cube or cube-connected-cycles. The pairing of

processors in such topologies is quite simple, a distributed scheme for the allocation of successive iterations is described in section 5.4.

In the occurrence of a hard failure, a processor might not output any result. This situation can easily be detected by the other processor in the pair by using a time-out mechanism. A well designed time-out interval would take into account any discrepancy in the execution time between the two processors or possible network delays.

5.3 Performance Evaluation

In this section, we derive the probabilities of the various outcomes at each iteration as well as the expected number of iterations. We will show that under normal conditions, these results compare favorably with those reported by Agrawal for the recursive algorithm [Agr85].

At each iteration of the algorithm, the possible outcomes are:

- *no-match*
- *match*, in which case two conditions are possible:
  - *correct result*
  - *incorrect result*

The following quantities are defined for a given task executing on a given processor. Let p be the probability that a task executes correctly; n the number of possible failure modes and q_i the probability of a failure in mode i, where $i = 1 \ldots n$. Therefore:

$$1 - p = \sum_{i=1}^{n} q_i$$

Let p_m be the probability of a match at a given iteration. Note that since the outcome at any iteration is independent of the outcomes at the previous iterations, p_m is independent of the number of iterations. We can write:

$$p_m = p^2 + \sum_{i=1}^{n} {q_i}^2 \tag{5.3.30}$$

A match can occur when both processors have the correct result or the same failure mode with respect to that computation. For simplicity we assume that $q_i = q \quad \forall i = 1 \ldots n$, therefore:

$$p_m = p^2 + \frac{(1-p)^2}{n} \tag{5.3.31}$$

Let $P_m(I)$ denote the probability of a match at the I^{th} iteration and I_{exp} the expected number of iterations, then:

$$P_m(I) = p_m(1 - p_m)^{(I-1)}$$

$$I_{exp} = \sum_{I=1}^{\infty} I P_m(I) = \sum_{I=1}^{\infty} I p_m(1 - p_m)^{(I-1)} = \frac{1}{p_m}$$

The quality of the decision made at any iteration is denoted by QD which is the conditional probability of a correct match given that a match occurred, therefore:

$$QD = \frac{p^2}{p_m} = \frac{p^2}{p^2 + \frac{(1-p)^2}{n}}$$

The improvement in quality brought by the algorithm over a non-redundant execution is evaluated by the Quality of Decision Improvement Factor ($QDIF$) which is defined as:

$$QDIF = \frac{1-p}{1-QD}$$

Where $(1-p)$ is the probability of an incorrect result on one processor and $(1-QD)$ is the probability of a match with incorrect results. Replacing with the expression for QD we obtain:

$$QDIF = n\frac{p^2}{1-p} + (1-p)$$

for large p ($p > 0.5$) $QDIF$ can be approximated by:

$$QDIF \approx n\frac{p^2}{1-p}$$

Figure 5.3 shows the plots of I_{exp} against p for different values of n. It can be observed that for $p \geq 0.3$ the value of n practically no effects on I_{exp}. This implies that I_{exp} has a low sensitivity to n and therefore the performance of the system is essentially independent of the number of possible failure modes.

Figure 5.4 shows the plots of QD against p for various n. As would be expected, the quality of the decision is superior for $n = 1000$. This is due to the fact that the number of failure modes increasing decreases the probability of an incorrect match. The same effect can be noticed in Figure 5.5 where $QDIF$ is plotted as a function of n and p. This figure shows $QDIF$ to be an exponentially increasing function of p.

Figure 5.6 shows a comparison of the expected number of trials in the iterative and recursive algorithms for $n = 100$. It demonstrates that for $p < 0.5$ the recursive algorithm is superior to the iterative one. However, under normal conditions, it can safely be assumed that $p > 0.5$, in which case the two algorithms have comparable performances. Since, in the recursive algorithm, the result at each iteration is checked against *all* the previous outcomes, the probability of a *match on incorrect result* increases with the number of iterations. Therefore, for low values of p, the expected number of iterations in the recursive algorithm is low because of the increased probability of a match on incorrect results.

In the iterative algorithm, the probability of a match is independent of the number of iterations which results in a large I_{exp} for low values of p. This characteristic has the effect of *reducing* the probability of an incorrect match when p is low given a maximum number of iterations possible. The maximum number of iterations possible is determined by the number of available processor pairs in the system. The iterative algorithm has the advantage of being less costly in execution (no reliance on a history of previous outcomes) and more suitable for a distributed environment. Note that the expression for QD is independent of the number of iterations unlike the recursive algorithm where the probability of a match at each trial depends on the number of preceding trials.

In Figure 5.7, we compare the quality of the decision in the pair-wise iterative algorithm to that in a similar TMR scheme. The envisioned TMR scheme is similar to the proposed dual redundancy scheme. It consists in executing each task on three processors, instead of two, and producing a result which would be the majority of the three outcomes.

- execute the task on two processors;
- increment *job-counter* in both processors;
- compare the two sets of results;
- if *match* then
  - send results to next tasks;
  - save copy of results as *checkpoint data*;
  - send *ack* messages to the processors where preceding tasks have executed;

  $\Longrightarrow$ preceding *checkpoint data* is deleted;
- else
  - increment *iteration-number* for the task;
  - increment *error-counter* in both processors;
  - send *negative ack* to preceding processors;

  $\Longrightarrow$ task is retried on a different pair of processors;

Figure 5.2: Iterative Distributed Algorithm

	QD, $n = 100$	
p	*Iterative*	*TMR*
0.5	0.9901	0.7952
0.6	0.9956	0.9064
0.7	0.9982	0.9645
0.8	0.9994	0.9901
0.85	0.9997	0.9958
0.9	0.99987	0.9987
0.95	0.99997	0.99980

Table 5.1: QD in the iterative and TMR schemes ($n = 100$).

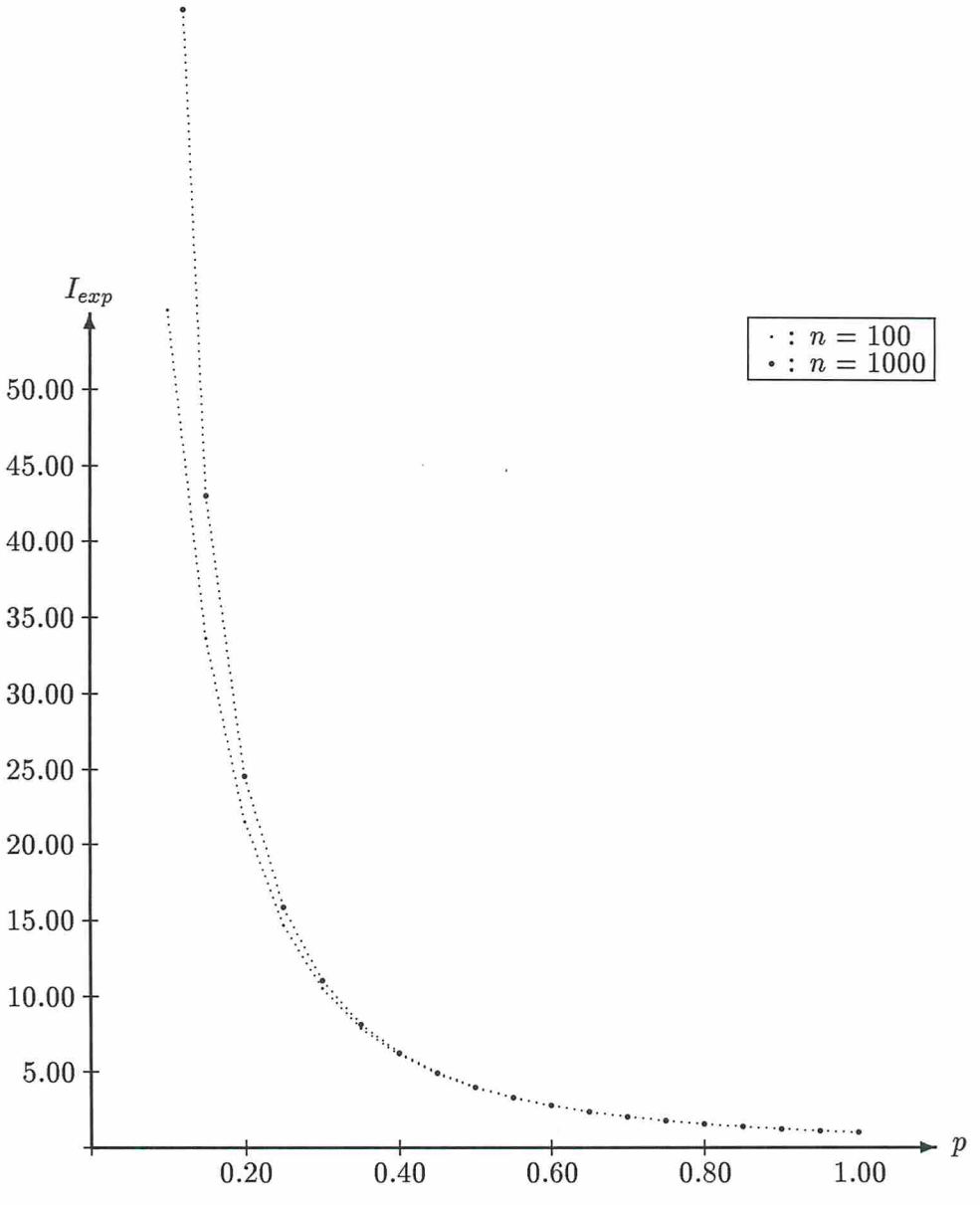


Figure 5.3: Expected number of iterations I_{exp} as function of n and p.

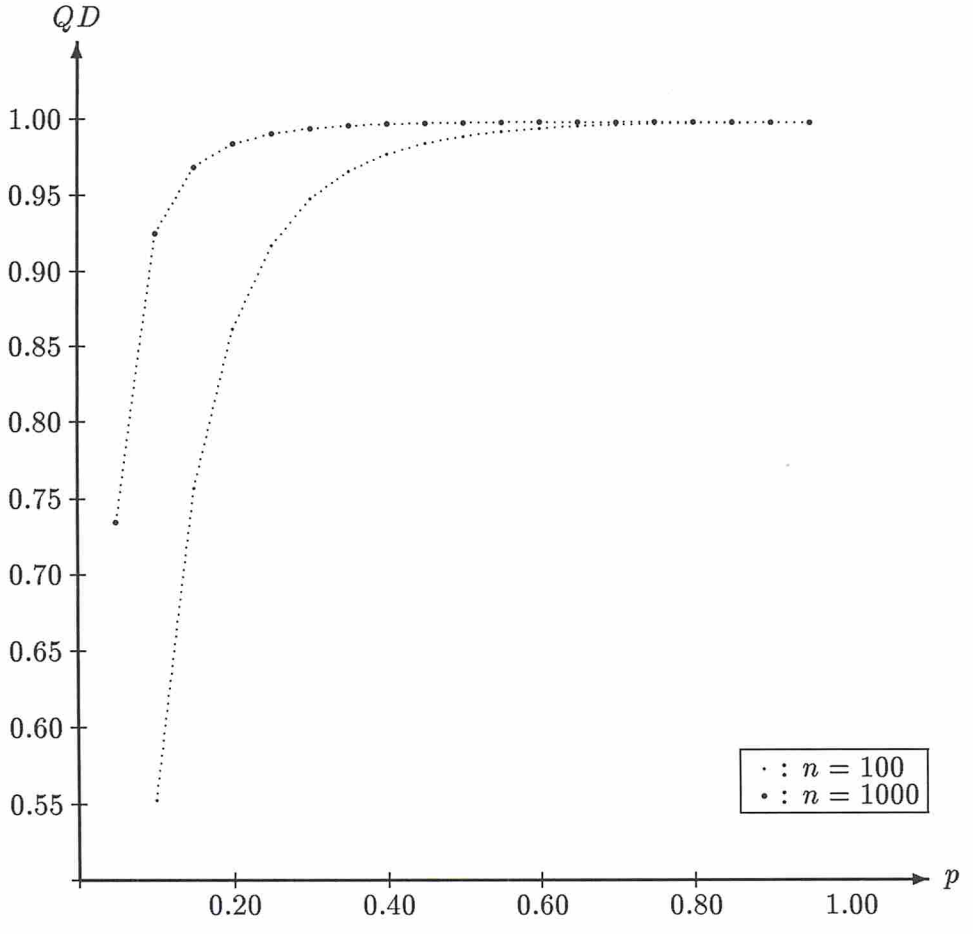


Figure 5.4: Quality of the decision QD as function of n and p

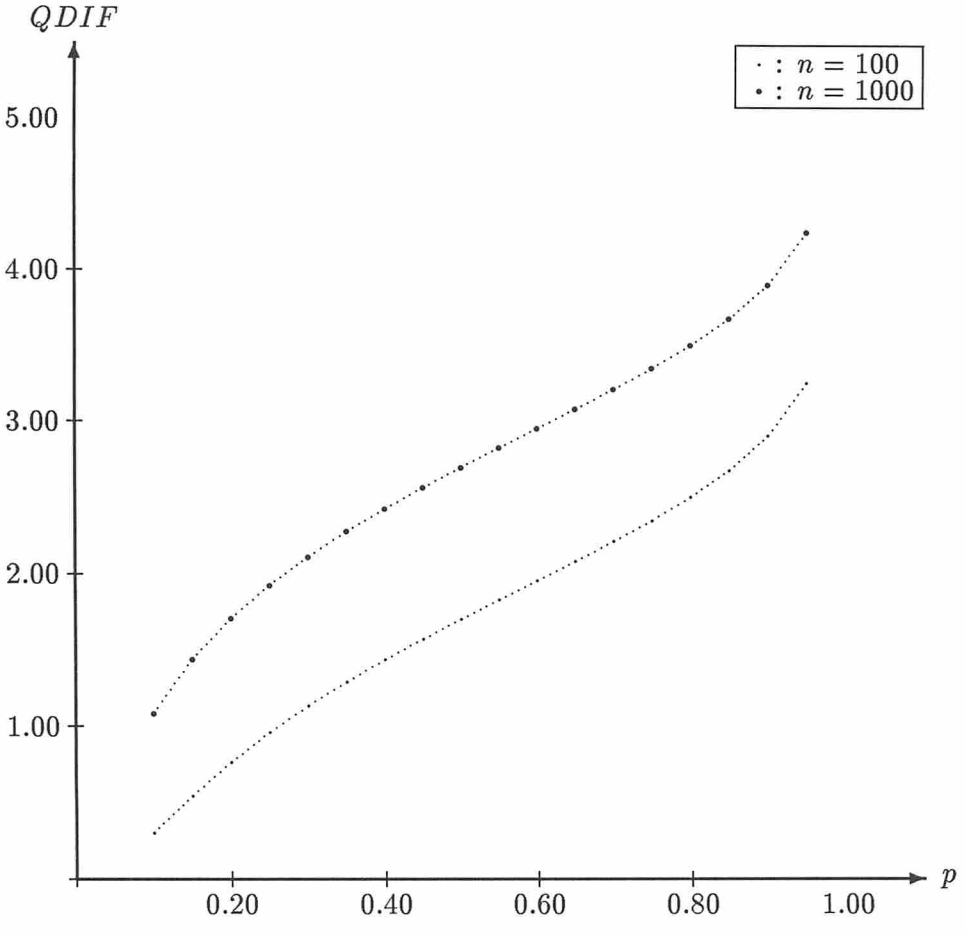


Figure 5.5: $QDIF$ as function of n and p

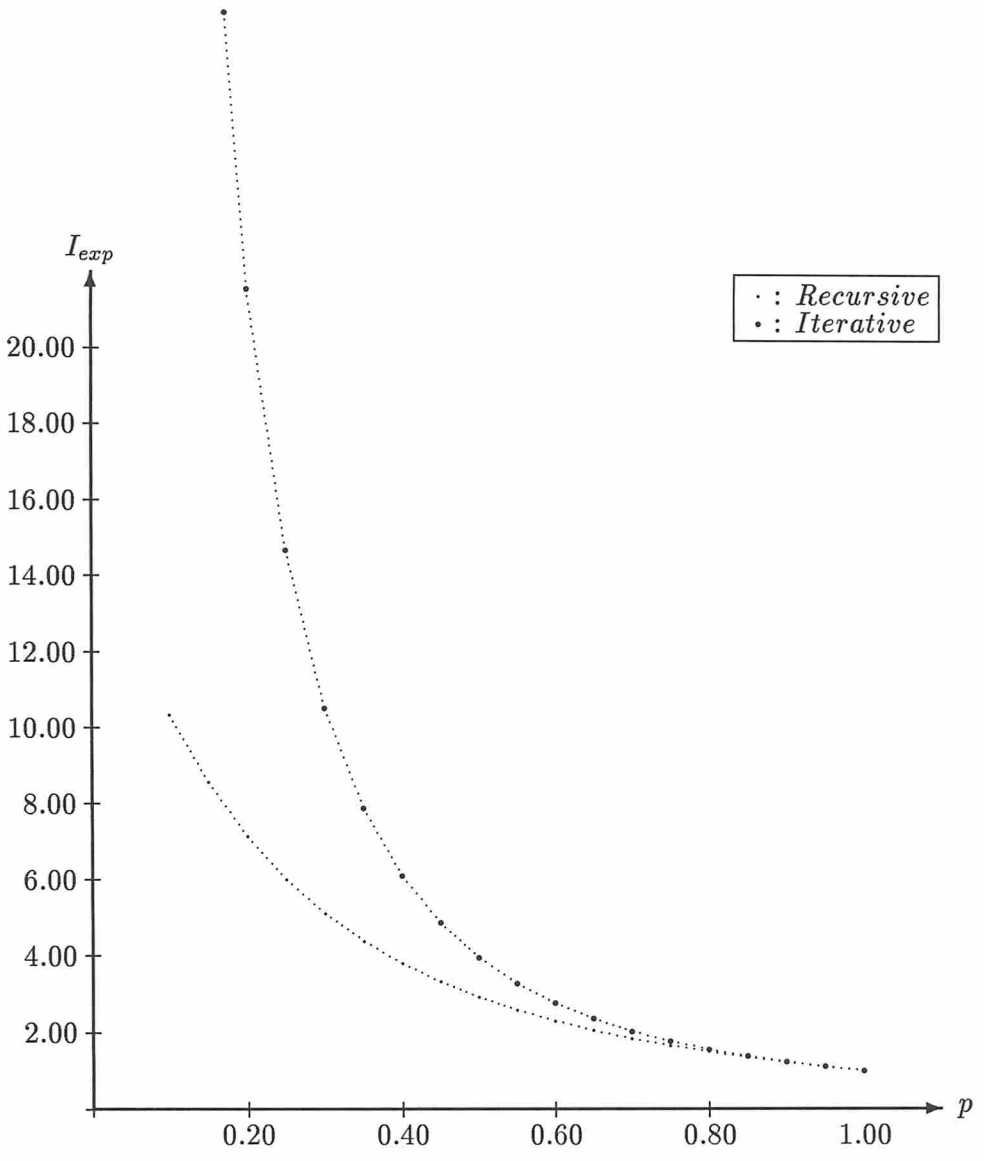


Figure 5.6: Comparison of the expected number of iterations for the iterative and recursive algorithms ($n = 100$)

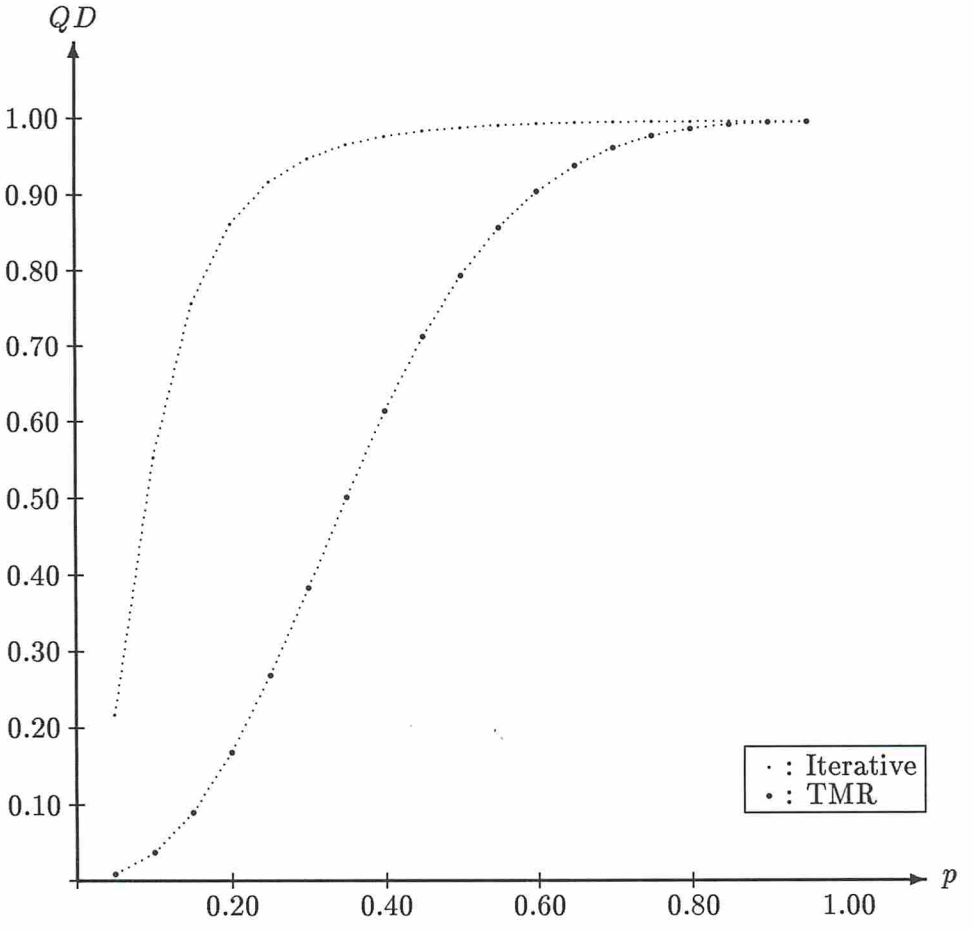


Figure 5.7: Comparison of QD in the iterative and TMR schemes, ($n = 100$).

The probability of a match in a TMR scheme can be derived as:

$$\begin{aligned} p_{tmr} &= p^3 + 3p^2 \sum_{i=1}^{n} q_i + 3p \sum_{i=1}^{n} q_i^2 + \sum_{i=1}^{n} q_i^3 \\ &= p^3 + 3p^2(1-p) + 3p\frac{(1-p)^2}{n} + \frac{(1-p)^3}{n^2} \end{aligned}$$

Therefore the quality of the decision is given by:

$$QD_{tmr} = \frac{p^3 + 3p^2(1-p)}{p_{tmr}}$$

Table 5.1 reproduces, numerically, these same results. It can be seen that even for very large values of p, the pair-wise scheme is substantially more effective than TMR.

The plots in Figure 5.7 show the quality of the decision in the two processors case to be superior to that of a TMR. The explanation of this counter-intuitive observation is similar to the discussion related to the recursive algorithm. As the number of trials is increased, the probability of an incorrect match increases, therefore reducing the quality of the decision. Note that the results in the pair-wise algorithm is a *unanimity* result, while in the TMR case it is a *majority* result and in the recursive algorithm it is a *plurality* result.

5.4 System Level Diagnostics

The purpose of a fault-tolerant algorithm is to protect the computation against element failures and to identify the failed elements. In this section, we discuss the potentials of the iterative algorithm for system level diagnostics.

In the description of the algorithm presented in Figure 5.2, an *error-counter* variable is incremented at every mismatch. Similarly, a *job-counter* variable is incremented for each task executed. These variables correspond to special registers in each processor. Algorithms that use these two variables for system level diagnostics are, conceptually, very similar to page replacement algorithms in virtual memory. However, the objective here is not to eliminate the most frequently failed processor, but to identify those processors that have a frequency of failures larger that an acceptable level. Whenever that level is reached in a processor, a signal is sent to all, or a subset of, its neighbors. One or more of these runs diagnostic tests on the suspected processor. If the processor passes the test, the counters are reset. This means that no hard failure exists in that processor and that the mismatchs occurred either because of transient failures or because of failures, permanent or transient, in the processor(s) it paired up with. If the test fails, its neighbors are notified of its demise and no future tasks are scheduled on that processor which results in its isolation from the system.

Note that the pairing of processors for task execution could either be *static* (i.e., a pair is formed by the same two processors) or *dynamic* (i.e., pairs are dynamically formed for each task execution). In either case, the two processors in pair do not have an identical set of neighbors which makes the testing more reliable. When static pairing is implemented, the demise of a processor implies the demise of its pair.

The scheduling of a task from one pair of processors to another amounts to implementing a distributed counting algorithm. Well known algorithms using Gray codes exist for popular topologies such as the hypercube.

Chapter 6

CONCLUSIONS

A dozen units had been pulled out, yet thanks to the multiple redundancy of its design ... the computer was still holding its own.

"Dave," said Hal, "I don't understand why you're doing this to me ... You are destroying my mind ... I will become childish .. I will become nothing ..."

Arthur C. Clarke
2001: A Space Odyssey

The general topic of this dissertation, as stated in Chapter 2, has been the identification and analysis of the major relevant issues in the fault-tolerant design and performance and reliability evaluation of massively parallel computing systems. Within the general scope of this subject area, we have identified and addressed in this dissertation three major topics.

- *Network Fault-Tolerance.*
  In Chapter 3 we have identified the effects of the network topology on the system connectivity and the ensuing probability of a network disconnection as a result of multiple failures.

- *Computational Reliability*
  The effects of an increased number of processors and on the trade-offs between a more powerful system and a higher failure rate were addressed in Chapter 4.

- *Distributed Fault-Tolerance*
  In Chapter 5 we have demonstrated that a functional execution model can allow for a very simple and inexpensive mechanism for run-time checkpointing and recovery. A distributed algorithm has been proposed that implements these functions and allows for system level diagnosis.

In this chapter we review and summarize the most relevant results presented in this dissertation along these three topics. Finally, we present a few concluding remarks on the issue of the fault-tolerance in massively parallel systems and propose directions for future research.

6.1 Summary of Results

In this section we present a summary of the major results that have been researched and described in this dissertation. In doing so we will also demonstrate the relevance and interdependence of the three main topics that have been addressed.

6.1.1 Network Fault-Tolerance

An analysis of the effects of multiple node failures on the network topology and its connectivity was presented in Chapter 3. This analysis focuses, in particular, on the probability of occurrence of a network partition as a result of these failures. The analysis, initially, addressed a family of regular graph topology, it was subsequently extended to non-regular graphs.

We have demonstrated, both analytically and using a Monte-Carlo simulation approach, that the case of a single node being cut out from the rest of the network, as a result of multiple nodes failing, is by far the most probable occurrence as opposed to a cluster of nodes being disconnected. These results also show that for a large number of processors, the frequency of a single node disconnection becomes very large (larger than 90%) and therefore the overall disconnection probability could be approximated by the single node disconnection probability.

Using these results, we were able to construct an analytical model of the disconnection probability based on the single node disconnection approximation. This model, in turn, was validated by showing it to fit closely the obtained simulation results. This model shows that the peak value of the probability of disconnection is mostly a function of the number of nodes, while the occurrence of this peak value is determined by the network connectivity.

In order to evaluate the potential for disconnections in a given network and compare various networks, we have introduced the measure of *Network Resilience*, $(NR(p))$, which is the number of failures a network can sustain while remaining connected with a probability $(1-p)$. Given a certainty level of no disconnection, as expressed by $(1-p)$, network resilience is the number of nodes that can fail without disconnection. Since a state of network disconnection would prohibit any recovery and therefore graceful degradation, the network resilience can be viewed as an evaluation of the number of allowable degradation states.

Evaluating the network resilience of various graph topologies shows that the resilience increases with an increase in the system size. However, when the connectivity is kept constant as in the torus and cube-connected cycles, the ratio of the network resilience to the total number of nodes, $(NR(p)/N)$, decreases. This implies that, for constant connectivity graphs, the number of degradation states would be a decreasing percentage of the number of nodes as the system size increases. For graphs where the connectivity increases with N, as in the case of the binary n-cube, the ratio increases as N is increased resulting in a number of degradation states that is an increasing fraction of N.

The disconnection analysis was extended to non-regular graphs such as the array. Simulation results show that the single node disconnection approximation is also valid for such graphs. However, these graphs were shown to have a lower network resilience than similar graphs of the same size. Addressing the issue of node disconnection due to link failures we show that links can be made to be far more fault-tolerant than nodes and therefore that link failures has a lesser relevance to the network partitioning problem. In this case also, simulation results of various networks show the single node disconnection to be by far the most frequent event due to link failures.

These results indicate that unless the connectivity is increased with N, the disconnection problem becomes very significant for large-scale systems. However, there are physical limitations on how much a node connectivity can be increased. Further, there are clear economical advantages in using low connectivity graphs. Therefore, low connectivity large-scale systems must be protected against the effects of node disconnections. One possible protection scheme consists in implementing redundant computations on redundant data as outlined in section 5.2. This scheme provides a continuously updated checkpoint that would allow the recovery a procedure in the case of a single node disconnection.

6.1.2 Computational Reliability

For large-scale systems a trade-off exists between a higher computing power and a higher failure rate of the system. In Chapter 4 we have analyzed the effects of an increase in the number of processors on the reliability of the system using a time-based analysis and the effects on the performance of the system were derived using a computation-based analysis. The analysis assumed a gracefully degradable system where the probability of successful recovery is expressed by a coverage factor c.

The time-based analysis addressed first the effects of the coverage factor and the system size on the mean time to failure. It was shown that the $MTTF$ is practically constant for a sufficiently large number of processors N, while for small values of N, the $MTTF$ is mostly insensitive to variations in c. Furthermore, it was shown that a substantial increase in the coverage factor results in a moderate increase in the $MTTF$. Therefore, for all practical purposes, the expected time to failure of a large-scale system is largely independent of N and is weakly affected by the value of the coverage factor. Second, the analysis addressed the effects on the mission-time (MT). The results, both analytical and numerical, show that the mission-time is maximal for a value of N denoted by N_p a decreases for an increasing $N > N_p$. It was demonstrated that the value of MT for $N > N_p$ is inversely proportional to both N and $(1 - c)$. The implications are that the time interval where a minimum reliability level is maintained (MT) decreases as the system size (N) is increased and that an improvement in the probability of successful recovery $(1 - c)$ is proportionately reflected in the value of MT. Therefore, for very large systems, the expected length of time where the reliability is maintained above a given minimum level becomes a small fraction of the single processor MTTF.

Analyzing the effects of c on the measure of computational work we show that there exists an upper-bound on the amount of computational work, expressed in processor-hours, a gracefully degradable system can deliver. Further, we demonstrate that the value of this upper-bound is independent of the initial number of processors and is a function of the coverage factor c and the single processor MTTF.

We introduced the measure of reliable processor-hours, RPH, which is the work delivered during the mission time interval. It was demonstrated that, for linear speed-up computations, RPH becomes constant for $N > N_p$. However, for a computation that exhibits sub-linear speed-up, the amount of *effective reliable computational work* decreases as $N > N_p$ increases. Therefore, for such conditions, the probability of a reliable completion of a computation is a decreasing function of N. This implies that, for sub-linear speed-up computations, an increase in the system size results in a decrease in the computational performability of the system unless the quality of the recovery algorithm is increased accordingly. This demonstrates that large-scale gracefully degradable systems do not scale up for $N > N_p$.

6.1.3 Distributed Fault-Tolerance

The problem of implementing a distributed fault-tolerance algorithm was addressed in Chapter 5. We have described a model for functional execution, that does not imply the execution of a functional language, but is based on the message-driven execution of tasks in a distributed system. In this model, we have demonstrated a run-time checkpointing mechanism that allows for a consistent global state to be saved in the system at all times and continuously updated. This, in turn, allows a simple recovery mechanism whenever a failure is detected.

Based on this mechanism, we have presented an iterative distributed fault-tolerant algorithm that relies on dual redundancy and the pair-wise comparison of results. The objectives of this

algorithm is two fold: (1) to provide a fault protection and therefore a more reliable execution and (2) to allow the detection of failures and hence provide for a system level diagnosis.

At each iteration, two processors are involved in the computation, independently and asynchronously. The outcome, at each iteration, is independent of the previous iterations. An iteration is repeated whenever a mismatch occurs. This algorithm does not rely on any central resource or data-structure and is therefore suitable for implementation on a distributed systems.

A probabilistic performance evaluation of the algorithm, based on multiple failure modes, shows it to be effective in providing fault-tolerant execution and run-time fault-detection. Its efficiency in providing correct results in the presence of faults was shown to superior to that of a similar TMR scheme while at two thirds of the hardware cost. Its overall performance was shown to compare favorably with that of a similar recursive algorithm (RAFT) with an increase in the hardware overhead. Unlike its recursive counterpart, the proposed iterative algorithm does not rely on shared resources. The proposed scheme is likely to provide effective fault protection in large-scale distributed systems where the large number of processing elements increases the expected failure rate by exploiting the increase in the available redundancy.

6.2 Future Research

This dissertation has addressed a number of issues in the fault-tolerant design and reliability and performance analysis of large-scale systems. It also provides a starting point for further work in this area. Some of the possible directions of future research are described in this section.

6.2.1 Communication Load

In this dissertation we have looked at the effects of failures on the reliability and computing performance of the system. Another effect of failures is a decrease in the overall communication bandwidth of the system. A state of network disconnection implies that no communication is possible between two sets of processors. Before that state is reached, however, the communication bandwidth would have decreased drastically. Two types of phenomena are conceivable when the communication structure is deformed:

- *Bottlenecks.*
- *Network Saturation.*

A bottleneck occurs when the communication bandwidth between two sets of processors is reduced to a very small value. The limit case being when a single link exists between two sets of processors. Bottlenecks can still occur when more than one links exists if the reduction in bandwidth is large enough.

A communication bottleneck can have very severe consequences on the performance of the system and could bring the system to a halt when no computational progress is being made. It is conceivable therefore that the system would fail by degrading to a level of unacceptable performance.

Network saturation can be seen as a mirror image of a bottleneck. When a node or a link fail its communication load is repartitioned, by the routing algorithm, among a number of alternative routes thereby increasing the communication load on several links and nodes. The occurrence of multiple failures could result in high communication loads on several segments in the system. This situation is akin to the "hot-spots" memory contention phenomena as described in [PN85]. Implementing an adaptive routing algorithm to balance the communication load throughout the

system would delay the occurrence of network saturation but would also spread it throughout the system.

6.2.2 Failure Rates

Previous research on the types and nature of faults had shown that the most frequent are transient faults. The total amount of transient faults in a system during a time interval is directly proportional to its size as measured by the number of switching devices or transistors. In a uniprocessor machine most of the switching elements are in the memory, and since only one memory location is accessed at any one time, the only one transient faults could be detected at a time. In a massively parallel machine, the ratio of processors size to memory size is substantially larger. Therefore:

- more transient failures could be experienced in the processing circuitry than in a uniprocessor machine; and
- more failures in memory could be detected at any one time.

These considerations imply that massively parallel machines are likely to experience a higher rate of transient failures due to (1) their size and (2) the inherent parallelism.

New experimental research is therefore needed to evaluate the rate of transient failures in massively parallel machines and assess their impact on the performance of the system.

6.2.3 Hardware Support

We have demonstrated, in this dissertation, that fault-tolerant design is a very critical issue in the development of large-scale systems. It is important, therefore, that the hardware design of such systems includes provisions for supporting fault-tolerance features such as:

- *Self-diagnosis.* An ability which would allow each processor to test itself and report its status to the system or a set of neighboring processors.
- *Mutual testing.* Which would allow processors to diagnose each other according to some predetermined algorithm. A consensus must then be reached as to the status of each processor and failed processors would be eliminated from the system.
- *Quiet shutdown.* Which would insure that a failed processor can be quietly shutdown and would not interfere with the operation of the system.
- *Checkpointing support.* Any checkpointing scheme eventually requires a fast and reliable access to I/O devices.
- *Reconfiguration.* Is necessary in situations where the network topology is of importance to the target application.
- *Backup or Redundant Data-paths.* These would help the system recover from a situation of bottleneck, network saturation or even disconnection.

Simplicity is a general requirement on any form of hardware support for fault-tolerance. The reason being that these hardware features can become a single point of failure in the system. Reducing their complexity, therefore, reduces their potential for failure.

6.2.4 Software Support

While extensive research has been done on various types of hardware fault-tolerant designs, little has been done on software supported fault-tolerance. In Chapter 5 we have demonstrated that a functional execution model could be used to implement a run-time checkpointing scheme. More research, however, is needed on fault-tolerant operating systems features such as distributed load-balancing and run-time system diagnosis. The availability of these features is an important factor in the development of gracefully degradable systems. In fact, graceful degradation cannot be achieved without support from the operating system which would have to logically reconfigure the system and insure a fair allocation of tasks to processors.

As massively parallel systems develop and become more accessible it is expected that more demand would be put on software systems and programming environments that would be suited for such systems.

References

[A*71] A. Avizienis et al. The STAR (self-testing and repairing) computer: an investigation of the theory and practice of fault-tolerant computer design. *IEEE Transactions on Computers*, C-20(10):1312–1321, October 1971.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[AD79] W.B. Ackerman and J.B. Dennis. *VAL-A Value-Oriented Algorithmic Language, Preliminary Reference Manual.* Technical Report TR-218, Laboratory for Computer Science, MIT, June 1979.

[Agh85] G.A. Agha. *Actors: A Model of Concurrent Computations in Distributed Systems.* Technical Report TR 884, MIT Artificial Intelligence Laboratory, June 1985.

[Agr85] P. Agrawal. RAFT: A recursive algorithm for fault-tolerance. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 814–821, 1985.

[Bac78] J. Backus. Can programming be liberated from the von Neuman style? *Communications of the ACM*, 21(8):613–641, 1978.

[Bat80] K.E. Batcher. Design of a massively parallel processor. *IEEE Transactions on Computers*, C-29(9):836–840, September 1980.

[BBN85a] *Butterfly (TM) Parallel Processor Overview.* Bolt Beranek and Newman Inc., Cambridge, MA, June 1985.

[BBN85b] *The Uniform System Approach To Programming the Butterfly (TM).* Bolt Beranek and Newman Inc., Cambridge, MA, November 1985.

[BCJ*71] W.G. Bouricius, W.C. Carter, D.C Jessep, P.R. Schneider, and A.B. Wadia. Reliability modeling for fault-tolerant computers. *IEEE Transactions on Computers*, C-20(11):1306–1311, November 1971.

[Bea78] M.D. Beaudry. Performance-related reliability measures for computing systems. *IEEE Transactions on Computers*, C-27(6):540–547, June 1978.

[BF76] M.A. Breuer and A.D. Friedman. *Diagnosis and Reliable Design of Digital Systems.* Computer Science Press, Rockville MD., 1976.

[CH81] K-Y. Chwa and L. Hakimi. Scheme for fault-tolerant computing: a comparison of modularly redundant and t-diagnosable systems. *Information and Control*, 49:212–238, June 1981.

[DB87] L. Donatello and Iyer B.R. Analysis of a composite performance reliability measure for fault-tolerant systems. *Journal of the ACM*, 34(1):179–199, January 1987.

[EH85] A-H. Esfahanian and L. Hakimi. Fault-tolerant routing in DeBruijn communication networks. *IEEE Transactions on Computers*, C-34(9):777–788, September 1985.

[FAH83] W.K. Fuchs, J.A. Abraham, and K-H. Huang. Concurrent error detection in VLSI interconnection networks. In *Proceedings of the* 10^{th} *Annual Symposium on Computer Architecture*, pages 309–315, 1983.

[Fen81] T-y. Feng. Survey of interconnection networks. *IEEE Computer*, 14(12):12–27, December 1981.

[FM84] D. G. Furchtgott and J. F. Meyer. A performability solution method for degradable nonrepairable systems. *IEEE Transactions on Computers*, C-33(6), June 1984.

[FR85] J.A.B. Fortes and C.S. Raghavendra. Gracefully degradable processor arrays. *IEEE Transactions on Computers*, C-34(11):1033–1044, November 1985.

[GR85] J-L. Gaudiot and C.S. Raghavendra. Fault-tolerance and data-flow systems. In *Proceedings of the* 5^{th} *International Conference on Distributed Computing Systems*, May 1985.

[HG87] K. Hwang and J. Ghosh. Hypernet: A communication-efficient architecture for constructing massively parallel computers. *IEEE Transactions on Computers*, C-36(12), December 1987.

[Hil86] D. Hillis. *The Connection Machine.* MIT Press, 1986.

[HSL78] A.L. Hopkins, T.B. Smith, and J.H. Lala. FTMP-A highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

[Kow85] J.S. Kowalik, editor. *Parallel MIMD Computation: HEP Supercomputing and its Applications.* The MIT Press, 1985.

[KR80] J.G. Kuhl and S.M. Reddy. Distributed fault-tolerance for large multiprocessor systems. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 23–30, July 1980.

[KR81] J.G. Kuhl and S.M. Reddy. Fault-diagnosis in fully distributed systems. In *Proceedings of the 11th International Symposium on Fault-Tolerant Computing*, June 1981.

[KR86] J.G. Kuhl and S.M. Reddy. Fault-tolerance considerations in large multiple-processor systems. *IEEE Computer*, 19(3):56–67, March 1986.

[LK86] F.C.H. Lin and R.M. Keller. Distributed recovery in applicative systems. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 405–412, August 1986.

[Mag80] G.A. Mago. A cellular computer architecture for functional programming. In *Proceedings of IEEE COMPCON*, 1980.

[Mal80] M. Malek. A comparison connection assignment for diagnosis of multiprocessor systems. In *Proceedings of the 7th Annual Symposium on Computer Architecture*, pages 31–35, May 1980.

[Mey80] J.F. Meyer. On evaluating the performance of degradable computer systems. *IEEE Transactions on Computers*, C-29(8):720–731, August 1980.

[Mey82] J.F. Meyer. Closed-form solutions of performability. *IEEE Transactions on Computers*, C-31(7):648–657, July 1982.

[MM82] M. Malek and J. Maeng. Partitioning of large multicomputer systems for efficient fault diagnosis. In 12th *International Symposium on Fault-Tolerant Computing*, pages 341–348, June 1982.

[MSA*85] J.R. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J.R.W Glauert, I. Dobes, and P. Hohensee. *SISAL-Streams and Iterations in a Single Assignement Language, Language Reference Manual, Version 1.2.* Technical Report TR M-146, University of California - Lawrence Livermore Laboratory, March 1985.

[MST79] S.R. McConnel, D.P. Sieworek, and M.M. Tsao. The measurement and analysis of transient errors in digital computing systems. In *Proceedings of the 1979 International Symposium on Fault-Tolerant Computing*, pages 67–70, June 1979.

[NG87a] W. Najjar and J-L. Gaudiot. Distributed fault-tolerance in data-driven architectures. In *Proceedings of the 2nd International Conference on Supercomputing*, May 1987.

[NG87b] W. Najjar and J-L. Gaudiot. Reliability and performance modelling of hypercube-based multiprocessors. In *Proceedings of the 2nd International Workshop on Applied*

Mathematics and Performance Reliability Models of Computer/Communication Systems, Rome, Italy, May 1987.

[NG88] W. Najjar and J-L. Gaudiot. Network disconnection in distributed systems. In *Proceedings of the 8^{th} International Conference on Distributed Computing Systems*, San Jose, CA, June 1988.

[NPA86] R.S. Nikhil, K. Pingali, and Arvind. *Id Nouveau.* Technical Report Computations Structures Group Memo 265, Laboratory for Computer Science, MIT, Cambrige, Massachussett, July 1986.

[PF82] J.H. Patel and L.Y. Fung. Concurrent error detection in ALU's by recomputing with shifted operands. *IEEE Transactions on Computers*, C-31(7):589–595, July 1982.

[PMC67] F.P. Preparata, G. Metze, and R.T. Chien. On the connection assignement problem of diagnosable systems. *IEEE Transactions on Electronic Computers*, EC-16:848–854, December 1967.

[PN85] G.F. Pfister and V.A. Norton. Hot spot contention and multistage interconnection networks. In *Proceedings of the 1985 International Conference on Parallel Processing*, August 1985.

[Pra85a] D.K. Pradhan. Dynamically restructurable fault-tolerant processor network architectures. *IEEE Transactions on Computers*, C-34(5):434–447, May 1985.

[Pra85b] D.K. Pradhan. Fault-tolerant multiprocessor link and bus network architectures. *IEEE Transactions on Computers*, C-34(1):33–45, Jannuary 1985.

[PV81] F.P. Preparata and J. Vuillemin. The cube-connected cycles: a versatile network for parallel computation. *Communications of the ACM*, 24(5), May 1981.

[RAE84] C.S. Raghavendra, A. Avizienis, and M.D. Ercegovac. Fault-tolerance in binary tree architecture. *IEEE Transactions on Computers*, C-33(6), June 1984.

[Ran87] A. Ranade. How to emulate shared memory. In 28^{th} *Annual Symposium on Foundations of Computer Science*, pages 185–194, Los Angeles, CA, October 1987.

[Ren86] D.A. Rennels. On implementing fault-tolerance in binary hypercubes. In *Proceedings of the 1986 Symposium on Fault-Tolerant Computing*, pages 344–349, 1986.

[Sei85] C. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1), January 1985.

[Ser84] O. Serlin. Fault-tolerant systems in commercial applications. *IEEE Computer*, 17(8):19–30, August 1984.

[SS82] D.P. Sieworek and R.S. Swartz. *The Theory and Practice of Reliable System Design.* Digital Press, Bedford, Mass., 1982.

[SSB87] A. Sengupta, A. Sen, and S. Bandyopadhyay. On an optimal fault-tolerant multiprocessor network architecture. *IEEE Transactions on Computers*, C-36(5):619–623, May 1987.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[STR88] R.M. Smith, K.S. Trivedi, and A.V. Ramesh. Performability aanalysis: measures, an algorithm, and a case study. *IEEE Transactions on Computers*, 37(4):406–417, April 1988.

[Tri82] K.S. Trivedi. *Probability and Statistics with Reliability, Queueing and Computer Science Applications.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

[Tro78] W. N. Troy. Fault-tolerant design of local ESS processors. *Proceedings of the IEEE*, 66(10), October 1978.

[vN56] J. von Neuman. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C.E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98, Princeton University Press, Princeton, NJ, 1956.

[WF84] C. Wu and T. Feng. *Interconnection Networks for Parallel and Distributed Processing.* IEEE Computer Society Press, 1984.

[WLG*78] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. SIFT: design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

Appendix A

Simulation Program

The following is the actual *C* program that was used for the Monte-Carlo simulation of network disconnection. The example bellow is for a binary n-cube topology the programs for other topologies differ only in the construction of the network.

```
#include <stdio.h>
#include <math.h>
#define MAXINT 2147483647

short cube[1024][10], fail[1024], n, N;
short mark[1024], rt;

short pow_two(k)
/* returns an integer power of two of the argument */
short k;
{
short i, p=1;
for(i=0; i<k; i++){p = p*2;}
return p;
}

get_partner(m,l)
/* returns the node number of the neighbor of a given */
/* node m along a given dimesion l */
short m,l;
{
unsigned p,q;
p = pow_two(l);
q = m ^ p;
return q;
}

void build_cube(n)
/* build the connectivity graph of a binary n-cube in */
/* the array cube */
```

```
short n;
{
short i, j;
for(i=0; i<n; i++)
{
for(j=0; j<N; j++)
{cube[j][i] = get_partner(j,i);}
}
}

short get_rt()
/* get a node number of the root for a deapth-first */
/* search of the graph looks for the first non-failed node */
{
short i=0;
while (fail[i++] == 1);
return (i-1);
}

void DFS(k)
/* deapth-first search of the graph with root as */
/* node k, marks marks with 1 non-failed nodes that */
/* are traversed. */
short k;
{
short i, v;
for(i=0; i<n; i++)
{
v = cube[k][i];
if (fail[v] == 0 && mark[v] == 0){mark[v] = 2;}
}
for(i=0; i<n; i++)
{
v = cube[k][i];
if (mark[v] == 2) {mark[v] = 1; DFS(v);}
}
}

short check_conn()
/* checks for connected components, returns the */
/* number of unmarked non-failed nodes */
{
short i, num_disc=0;

for(i=0; i<N; i++){mark[i] = 0;}
rt = get_rt();
```

```
mark[rt] = 1;
DFS(rt);
for(i=0; i<N; i++){if (mark[i] == 0 && fail[i] == 0)
{num_disc++;}}
return num_disc;
}

short get_node(h)
/* gets next node to fail with uniform distribution */
short h;
{
long l;
double z;
short d, i, count=0;
l = random();
z = (double)l/MAXINT;
d = z*(N-h-1) +1;
for(i=0; i<N; i++)
{count = count + (1 - fail[i]);
  if (count == d) break; }
return i;
}

main(argc,argv)
int argc;
char *argv[];
{
FILE *fpo, *fopen();
short i, last, j, DISC, v, degree[1024], T, I, J, Iter;
double pd[1024], disc_size[1024];

if ((fpo = fopen(*++argv,"w")) == NULL)
{printf("cannot open file %s\n", *argv); exit(0);}

printf("enter n=");
scanf("%hd", &n);
N = pow_two(n);
printf("n= %hd and N= %hd\n",n,N);
build_cube(n);
printf("enter Iter=");
scanf("%hd", &Iter);
srandom(time(0));

for(I=0; I<Iter; I++)
{
for(i=0; i<N; i++){fail[i] = 0;}
```

```
for(i=0; i<N; i++){degree[i] = n;}
DISC = 0;
for(i=0; i<N; i++)
{
last = get_node(i);
fail[last] = 1;
for(j=0; j<n; j++)
{
v = cube[last][j];
degree[v] = degree[v] - 1;
if (degree[v] == 0 && fail[v] == 0)
{DISC = 1; break;}
}
if (DISC == 1) {T=i; break;}
if (i > 2*(n-2)) {
DISC = check_conn();}
if (DISC > 0) {T=i; break;}

}
pd[T]++; disc_size[DISC]++;
}
fprintf(fpo,"I Disc Prob\n");
fprintf(fpo,"\n");
for(i=0; i<N-1; i++)
{
pd[i] = pd[i]/((double)Iter);
fprintf(fpo,"%d %g\n",i+1,pd[i]);

}
fprintf(fpo,"\n");
fprintf(fpo,"Size Disc Prob\n");
fprintf(fpo,"\n");
for(i=0; i<N-1; i++)
{
disc_size[i] = disc_size[i]/((double)Iter);
if (disc_size[i] > 0)
{
  printf("size = %d and disc prob = %g\n",i,disc_size[i]);
  fprintf(fpo,"%d %g\n",i,disc_size[i]);
     }
}
}
```

MARY'S DISK

Name	Size	Kind	Last Modified
88-60	1K	Draw 1.95 document	Tue, Feb 14, 1989 12:36
89-01	2K	Draw 1.95 document	Tue, Feb 14, 1989 12:58
89-02	2K	Draw 1.95 document	Mon, Apr 24, 1989 14:33
89-03	1K	Draw 1.95 document	Fri, Feb 24, 1989 12:10
89-05	1K	Draw 1.95 document	Mon, May 22, 1989 9:32
89-06	2K	Draw 1.95 document	Tue, May 2, 1989 14:27
89-07	2K	Draw 1.95 document	Mon, May 22, 1989 9:28
89-15	1K	Draw 1.95 document	Wed, Jun 28, 1989 14:39
89-16	1K	Draw 1.95 document	Wed, Jun 28, 1989 14:52
89-17	1K	Draw 1.95 document	Wed, Jun 28, 1989 14:55
89-18	1K	Draw 1.95 document	Wed, Jun 28, 1989 15:06
89-20	1K	Draw 1.95 document	Fri, Jul 28, 1989 15:10
89-21	2K	Draw 1.95 document	Thu, Aug 3, 1989 14:03
89-22	1K	Draw 1.95 document	Tue, Aug 8, 1989 10:24
89-28	2K	Draw 1.95 document	Thu, Oct 12, 1989 13:26
CENG	2K	Microsoft Word d...	Wed, Jun 14, 1989 15:51
ceng 89-09	2K	Draw 1.95 document	Mon, Jun 5, 1989 8:28
cnr	2K	Draw 1.95 document	Thu, Dec 15, 1988 14:46
CRI 88-61	1K	Draw 1.95 document	Mon, Mar 13, 1989 17:44
cri2	2K	Draw 1.95 document	Thu, Jan 19, 1989 14:17
cri3	2K	Draw 1.95 document	Tue, Jan 17, 1989 12:43
crititle	2K	Draw 1.95 document	Thu, Dec 8, 1988 11:48
DISTRIBUTION	4K	MacWrite document	Mon, Feb 27, 1989 13:58
ee680	1K	Draw 1.95 document	Mon, Mar 6, 1989 14:29
eval	3K	Draw 1.95 document	Fri, Dec 2, 1988 14:39
eval2	2K	Draw 1.95 document	Thu, Dec 1, 1988 16:28
faculty list	3K	Microsoft Word d...	Thu, Jun 22, 1989 10:10
food list	3K	MacWrite document	Wed, Mar 22, 1989 11:24
MIT	1K	Draw 1.95 document	Wed, Dec 14, 1988 16:01
PARKER1	3K	Draw 1.95 document	Wed, Aug 16, 1989 15:38
PARKER2	2K	Draw 1.95 document	Wed, Aug 16, 1989 14:30
toole	2K	Draw 1.95 document	Wed, Dec 14, 1988 14:21