This is to certify that the

dissertation entitled

Reliable Parallel Processing
The Application Oriented Paradigm

presented by

Bruce Malcolm McMillin

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Major professor

Date June 20, 1988

RELIABLE PARALLEL PROCESSING
THE APPLICATION ORIENTED PARADIGM

A Dissertation

By

Bruce Malcolm McMillin

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1988

ABSTRACT

RELIABLE PARALLEL PROCESSING THE APPLICATION ORIENTED PARADIGM

By

Bruce Malcolm McMillin

Fault-tolerance is of paramount importance in large distributed multiprocessor systems. The key to providing reliability for this necessarily applications oriented environment, lies in low cost and easily programmable solutions. This motivates consideration of a new class of fault tolerance techniques - the *Application Oriented Paradigm.*

This dissertation provides both the basis for construction of the requisite application oriented constraint predicates and the necessary tools for a reliable consistent distributed diagnosis in the Byzantine failure environment. The approach taken here is novel in the sense that a *Constraint Predicate* is formulated from a basis set of design metrics which embody desirable facets of the solution theory. The predicate is then an application oriented abstraction of the necessary fault tolerance. Reliable consistent distributed diagnosis appears as part of the environment to the programmer. Both Byzantine Agreement and syndrome testing are treated as diagnostic techniques.

The result of this dichotomization of fault tolerance is a provable and usable environment for *Reliable Parallel Processing*.

To Patricia

ACKNOWLEDGEMENTS

I wish to thank the members of my Ph.D. committee for their guidance and support during my Ph.D. program. I would particularly like to express my appreciation to A. Wojcik for his leadership of the Department, to A. Esfahanian for lending an ear, and mostly to L. Ni for providing a nearly optimal environment for his Ph.D. students. I also wish to thank both Dr. Wojcik and Dr. Fox for their careful reading of this disseration and to Dr. Hughes for his insightful comments.

To my parents, Kenneth and Phyllis McMillin, I express my appreciation for the early encouragement they gave that enabled me to pursue this endeavor. I would like to acknowledge all the people that touched my life while I was at Michigan State. Too numerous to mention all, in particular, I do wish to acknowledge my good friends Ray and Atsuko Klein, Pat Flynn, Eric Wu, Yew-Huey Liu, Chung-Ta King, Youran Lan, and Taieb Znati.

To my wife Patty, for whom I hold the greatest love, no further words need be said.

This work was supported in part by the DARPA ACMP project, in part by the State Of Michigan RE/ED project, in part by a Michigan State University Graduate Office Fellowship, and in part by the GTE Graduate Research Fellowship Program.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Massively parallel *distributed-memory multiprocessors* (DMMP's) have gained much attention recently due to their unique scalability in providing massive computing power. Various hypercube multiprocessors, such as the Ncube, iPSC, and FPS T-series, are notable examples of commercially available DMMPs [ShFi88, Hwan87]. Design of parallel algorithms for DMMPs has been a major research issue in the application domain. However, the design of reliable parallel algorithms has received little attention. As the size of DMMPs grows into thousands of processors and the corresponding scale of attempted application problems grows even faster, solution to the problem of providing fault-tolerance is of paramount importance to the greater use of parallel processing.

Traditional fault-tolerance techniques are mainly geared to providing an ultra-reliable environment [HoSL78, Wens78, Lala86]. The ultra-reliable environment is necessary when loss of property or life is the result of a failure. DMMPs often operate under less stringent reliability requirements. However, a failure, while not catastrophic, may increase application turnaround time to unacceptable levels (reference Chapter 2). The number of components of a large scale DMMP makes it particularly susceptible to failure. To provide the appropriate level of reliability it is necessary to re-examine the notion of fault-tolerance and how to provide it.

Reliability should be treated as part of the operating environment much as the compilers, libraries, and operating system appear to the programmer. Ideally, reliability is entirely transparent to the programmer. However, the cost of techniques to furnish this

transparency can be prohibitive or the techniques themselves infeasible. Some of the existing techniques to provide the necessary reliability include the use of better components (*Fault Avoidance*), stopping the system when a fault occurs (*Fault Detection*), running in the presence of faults (*Masking Redundancy*) or removing failed components from the system (*Reconfiguration*). For DMMPs, which are constructed from "off the shelf" building blocks, the last two techniques are most applicable. As we shall see, these two techniques require communication and agreement among the components of the DMMP. This agreement is commonly obtained by *Voting*.

1.1 MODERN FAULT TOLERANCE

Early work in computer system fault tolerance was concerned with failures of individual components. Much of the work thus was concerned with gate level issues such as "stuck at faults" [KrTo81] and other low level issues in processor design. This was primarily owing to the single CPU employed and its expensive hardware components. In modern systems and particularly in massively parallel DMMP design, failed components are considered to be at the granularity of the processor, the memory, and the bus [Wens78]. This means that when a failure occurs in a component, we wish to isolate its effects on the remaining portion of the system. We are not concerned with how the component failed, simply that it did so. When a component fails, we expect to have enough other similar components to take its place. These components, in the interest of low cost, are typically off-the-shelf processor, disk, and memory designs. The desired reliability may not be built into these components. Thus we build the desired level of redundancy through system level algorithms to coordinate these components. Finally, as systems become more complex, low level analysis of fault-tolerance and low level redundancy design are not feasible.

1.2 BACKGROUND ON RELIABLE SYSTEMS

Early work in the area of computer system reliability was focused in the space program and in the telephone switching environment. The OAO satellite computer [Kueh69] used discrete component redundancy. This technique became outmoded, as with integrated circuits the probability of multiple transistors being destroyed is about the same as for a single transistor [Renn84]. The initial Bell ESS design (for telephone switching purposes) uses two way redundancy [Toyw78]. Two computers perform the application in parallel. If one computer failed, the independent results would disagree and the entire ESS would halt. The advantage of this approach is that an incorrect output is never produced. A more modern example of computer fault-tolerance is in the Apollo guidance system. This system used triple modular redundancy (TMR) in which three times the necessary resources are employed. Three processors run the same program and vote to mask errors in any single processor [AnMa67]. The JPL star is a computer designed for deep space exploration where human maintenance is impossible. It has a large number of functional units with spares which can be switched in automatically to replace a failing component [Aviz71]. These designs form a basis for more modern fault-tolerant architectures.

1.2.1 Fault Tolerant Multi-Processor (FTMP)

The *Fault-Tolerant Multi-Processor* (FTMP) [HoSL78] is an extension of early spacecraft designs. The basis for design of the FTMP is to mask all errors from the applications software. This is advantageous since no software rollbacks are required to fix errors. This masking is accomplished through a TMR design extension known as *Parallel Hybrid Redundancy*. A pool consisting of multiple triads of modules such as processors, memory, and busses comprise the system. Each triad has a voting element which votes on the data presented to it in triplicate. A module may be retired, regrouped into another triad, or assigned to perform maintenance diagnostic testing. A module is retired when it has been found to have failed at which time a spare is switched in its

place. If no spares are available, the triad is retired. Since the hardware resources are allocated from a pool, the application never feels the effects of these reconfiguration activities.

Voting at the hardware level is done bit by bit in the the tightly synchronized approach adopted. Synchronization is achieved through keeping the clocks of each autonomous processor in absolute synchrony. This is accomplished by the use of TMR communication between the clocks at the hardware level. The voting elements detect failures in a 2-of-3 vote. Another controller called a bus guardian isolates or reconfigures around the failing module. If a bus is faulty, the bus guardians are directed to switch to another bus. If a processor is found to be faulty, it is isolated from the bus by the low level logic of the bus guardian.

Voting is not the only means employed in the FTMP for fault-detection. Modules may be assigned in a maintenance mode to exercise the system to attempt to find failed components and to remove them as quickly as possible. This testing checks each processor against multiple busses to attempt to narrow down the actual location of the fault, either processor or bus.

The advantage of this tight synchronization, other than the ability to mask faults at the hardware level, is that no buffering, synchronization primitives, or completion intervals are required. This implies a speed improvement over a machine which employs loose synchrony. There are, however, disadvantages to this method of low level voting. A voter or bus guardian failure can cause a catastrophic hardware failure that will probably defy any method of software correction. The tight synchronization also has limitations that will be discussed in a following section.

1.2.2 Software Implemented Fault-Tolerance (SIFT)

A contrasting approach to the FTMP, using loose synchrony, is the *Software Implemented Fault-Tolerant* computer (SIFT) [Wens78]. The SIFT was designed to serve in a

similar fault tolerant capacity as FTMP. The SIFT uses multiple busses and modules, however, the modules (processors) are off-the-shelf components. The SIFT concept is currently employed in state of the art systems such as the space shuttle [Skla76].

The loose synchrony of the SIFT allows it to run as a loosely coupled multiprocessor. Non-critical applications can take advantage of this multiprocessor speedup. For critical applications, the SIFT can form into a redundant configuration using TMR redundancy. Fault isolation is obtained by physical isolation of failed hardware components. Fault masking is enhanced through the use of multiple busses over which multiple copies of the data are transmitted. Voting takes place as a function of the software. Critical tasks in the SIFT are required to be iterative tasks such that voting is done on the state of data before each iteration. This guarantees that each processor is working on the same loop. If an error is discovered in the vote on this state of the system, the software attempts to locate and remove the faulty component. This is done by changing the bus of the presumed faulty processor. If the fault follows the move, the processor is retired. If the fault disappears, the bus is retired. Thus the SIFT can configure around the fault. This detection, however, is not complete. Each processor makes a report of failed components to the runtime supervisor. Depending on the type of fault, it may not be decidable which processor or bus is actually failing. However, the system can continue to run in a fault masking mode.

The clocks in the SIFT are not synchronous. However, for reasons that will made clear in Chapter 5, the clocks must be kept in partial synchrony. With respect to the hardware, each processor has an independent clock. These clocks are resynchronized occasionally by an algorithm known as a clock skew reset. This algorithm is essentially a voting algorithm which requires at least four clocks to mask one fault [PeSL80].

Low level hardware correction is not employed in the modules of the SIFT. Through modeling work done by the system designers, only a slight increase in reliability may be gained by adding this hardware support over what is achieved through the voting.

The loose synchrony and high level error correction of the SIFT, as will be seen shortly, can provide a basis for the construction of reliable distributed systems.

1.2.3 Space Shuttle Computer Control System

The space shuttle computer control system is of an ultra-reliable design as are the FTMP and SIFT. The important difference is that the space shuttle computers are constrained to be off-the-shelf processing systems, specifically, IBM AP-101 CPUs [Skla76]. Thus the space shuttle design is consistent with the more modern notion of fault-tolerant systems outlined at the beginning of this chapter.

The hardware configuration of the space shuttle system consists of five computers. Four of the computers are programmed identically to perform flight critical functions such as guidance, navigation, and control. The outputs of the four computers are voted in the control actuators with the inputs to the calculations coming from multiple input sensors.

Faults are not simply masked. The flight crew must be notified of a failed computer. Furthermore, the faults are isolated; that is, one computer failure cannot cause another computer to erroneously identify itself as faulty. This allows the crew to reallocate resources such that the system many continue to operate in the presence of up to 3 failed computers. In the case where only two computers are left operational, fault identification is not possible; however, fault detection is possible in the sense of the Bell ESS systems. A fault occurrence can be detected but not located.

The fifth computer is programmed to the same set of specifications as the four primary computers, but is implemented by a different contractor [Renn84]. This allows detection of software design and coding errors thus implementing a degree of software error detection. In the case of a disagreement, the correct version of the software must be determined by the crew, and that version then taken as correct.

The space shuttle provides both hardware and software fault tolerance, continued operation in the presence of hardware failures, and detection of implementation problems in the software. As we shall see in Chapter 2, these two facets of fault-tolerance also form the basis for the application-oriented fault-tolerance paradigm described in this thesis.

1.3 FAULT TOLERANCE TECHNIQUES

The previous introductory examples have shown the origins of fault-tolerant computing design. A more rigorous treatment of these techniques is presented in this section.

System reliability can be achieved through the alternative event paths shown in Figure 1.1. [Siew82, KuRe86, YaHa84].

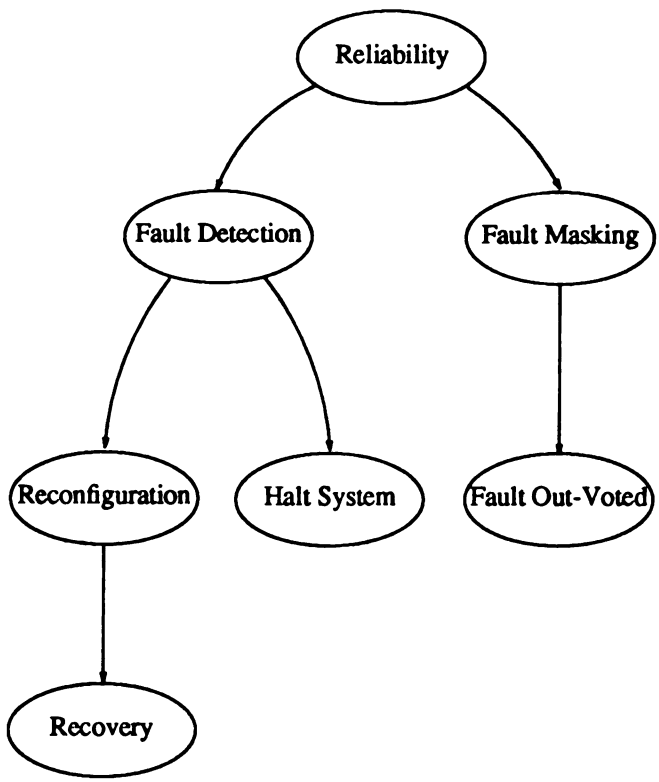

Figure 1.1. Alternative paths to achieve reliability

1.3.1 Fault Detection

Reliable fault detection informs a trusted observer or observers when a fault occurs. In a DMMP there may be no centralized observer, or use of a centralized controller may be prohibitive in terms of run time penalty. However, for hardware fault-tolerance, trusted observers may simply be a failed component's peers. If the number of faulty components in the system is a bounded design parameter, then sufficient non-faulty components always are available for reliable detection. In the distributed environment, during problem execution, it is not known by all components which components are faulty and which are reliable leading to a possibly inconsistent diagnosis.

The problem of reliable hardware fault detection is twofold: the achievement of a consistent diagnosis, and the achievement of a correct diagnosis though appropriate testing. Two possible techniques to achieve this consistent diagnosis are presented in Chapter 3.

Software fault detection, assuming a reliable hardware execution environment, need not be concerned with consensus on the location of a failed component. A single test/processor upon detection of a failed component can reliably signal the error instance.

In both the hardware and software failure environments, generation of an appropriate test is the common task.

1.3.2 Fault Masking

Reliable fault masking uses non-faulty elements to compensate for the effects of faulty elements. *N-Modular Redundancy* (NMR), a more general case of the TMR employed in the Apollo system, is a common technique to mask failures. The idea is to take a majority vote on a calculation replicated N times. This, of course, is expensive in terms of either hardware or run-time cost. A hardware solution requires N times the number of processors as well as a reliable voter. Software solutions require each processor to run N copies of surrounding computations and then vote on the result. This slows

down the computation by at least a factor of N. However, as described earlier, NMR required in ultra-reliable systems such as the Apollo spacecraft control, the FTMP and SIFT aircraft control systems, and the FTP reactor control system [Siew82, Hopk77, HoSL78, Renn84, Wens78, Lala86].

1.3.3 Reconfiguration

Reliable reconfiguration removes the faulty component from the system either logically or physically. Centralized reconfiguration control is the easiest to implement. However, this technique may be unacceptable in a distributed system because it introduces a single point of failure, namely at the controller. Decentralized reconfiguration control is more appealing but harder to implement. [Garc82] proposed the idea of a local coordinator to oversee the reconfiguration process. The local coordinator must be elected in a reliable election. This can be done via a voting process. The local coordinator then attempts to map in a spare component or to assign the job from the failed component to another functioning component.

When a hardware component fails, it may have suffered only a transient failure and can be reenstated into the system if the operation, when retried, succeeds. If the component is found to have suffered a permanent failure, then it can be replaced by an identical copy. Software failures are completely different. Since software design/coding errors are repeatable, once an error is detected, replacing the copy with another identical copy is futile since the same error will result. Thus the erroring algorithm must be replaced by an alternate algorithm. Unlike the hardware failure environment, for the software environment, it may be desirable to return to the primary algorithm after the failed problem instance has been completed by the alternate algorithm.

1.3.4 Recovery

Reliable recovery involves rolling back the computation to a suitable checkpoint and restarting from that point. A local coordinator may assume this function and initiate the rollback. The checkpoints may be held within the reliable components or held by the coordinator. Once the system is recovered, it ignores any messages from a failed hardware component.

1.4 RESOURCE FAILURE MODELS

The previous four types of fault-tolerance techniques are all a function of the particular fault model under consideration. Resource failures can be classified (ranging from most to least restrictive) by the classification given in Table 1.1. The relationship between them is shown in Figure 1.2.

1) Fail-Stop	A failing resource detects its own failure and stops. A Fail-Stop resource never generates spurious or incorrect messages. The resources connected to a stopped Fail-Stop resource immediately know that it has stopped. This is the easiest form of fault to detect, but is probably the hardest for the resource to implement.
2) Hard	The resource, when tested by a non-faulty testing unit, always responds with its correct status, faulty or non-faulty. This is the fault model assumption made in the PMC [PrMC67] model of fault diagnosis which will be discussed further under syndrome testing.
3) Intermittent	A resource, when tested by a non-faulty testing unit, may or may not respond correctly without restriction.
4) Byzantine	Byzantine resources encompass the universal class of faults. Any failure to meet specifications is a Byzantine fault. A Byzantine fault may generate incorrect or even malicious results to a test query or during algorithm participation.

Table 1.1. Fault Models

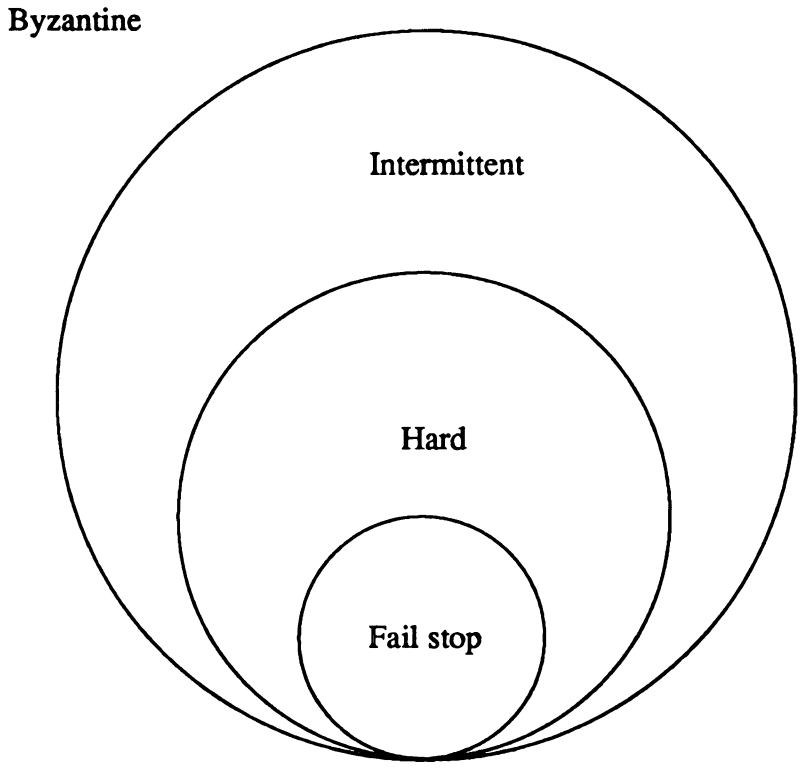

Figure 1.2. Fault Model Relationships

1.5 DISTRIBUTED MEMORY MULTIPROCESSORS (DMMPs)

A DMMP (Figure 1.3) is a system of N autonomous MIMD (Multiple Instruction Multiple Data Stream) processors each with their own local memory. These processors communicate with each other via message over an interconnect. This message passing takes non-negligible communication time. Examples of commercially available DMMPs were presented in the introduction to this chapter. These systems currently are on the order of thousands of processors. SIMD (Single Instruction Multiple Data Stream) architectures such as the Connection Machine [Hill85] have 65 thousand processors. It is not hard to make the transition to DMMPs of at least this scale.

The Intel iPSC and Ncube computers make a distinction between host processors and node processors. Both have an additional host processor which is used for program

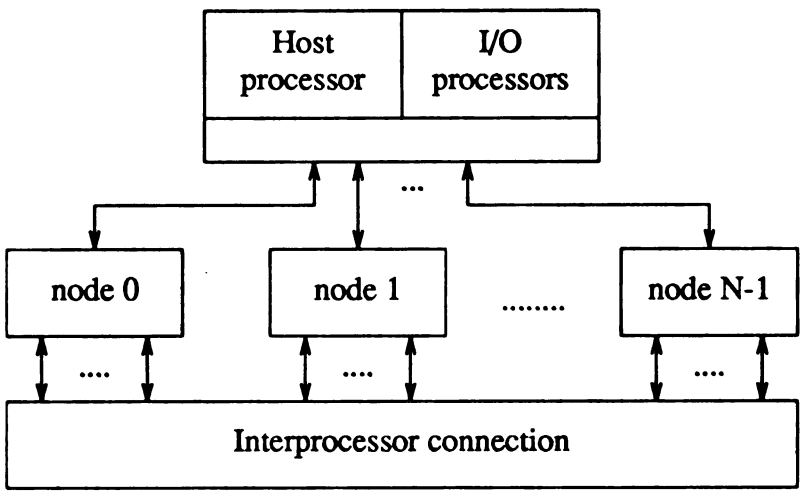

Figure 1.3. Distributed-Memory Multiprocessor System (DMMP) Architecture

downloading/uploading and data transfer. Note that while the host has a connection to each node, the number of nodes precludes the host processor from becoming involved in the nodes' calculations.

Design of a support environment requires consideration of the system characteristics. The primary issues of concern here are the locality of information and the nature of distributed control.

1.5.1 Locality of Information

The lack of shared data structures, or more precisely the limitation of a particular testing processor to examine another's internal memory, reduces testing capabilities. A test can only make a decision based on information from the local internal state of the computation and the messages it receives from other processors (peers) in the computation.

While privacy of information can be a drawback, it also enhances the fault-tolerance of a DMMP. In a faulty DMMP, corruption of the environment by a particular

processor is limited to the message it sends. Contrast this with the shared memory model of multiprocessing in which a processor can undetectably corrupt a computation at any time through disruption of the shared store. Thus correctness assertions need only consider the message exchange. This considerably enhances the tractability of the assertion development problem.

To categorize the amount of information available in the distributed environment, the terms "White Box" and "Black Box" are employed. These terms are used in the software engineering environment to dichotomize the two approaches coverage test generation. Here, in the white box environment, the participants in the distributed environment have perfect information. The complete internal state of a particular processor is known to other processors. This is consistent with the shared memory model of multiprocessing but is inconsistent, due to infeasibility, with the DMMP model. The black box environment is characterized by the true DMMP. Participants in the distributed environment have, as their only information, local state information and received messages.

1.5.2 Control

The large size of a DMMP dictates that operation under a centralized control scheme is infeasible due to the performance bottleneck in the central control unit. Furthermore, from a reliability standpoint, a centralized controller introduces the undesirable single point of failure. Thus, fostered by the locality of information and the size of DMMPs, individual processors function together under distributed control. The locality of information, coupled with non-negligible communication delays between the processors, results in a lack of state consistency for any particular snapshot of the system. Since processors in DMMPs are of the MIMD type, there may be no synchronization, or at best partial synchronization, between the units. Any algorithm to provide fault tolerance must be able to function in the presence of imperfect information, temporarily

inconsistent information, and under decentralized control.

1.6 THE APPLICATION ORIENTED FAULT-TOLERANCE PARADIGM

Out of the three problems of reliable fault detection, reconfiguration, and recovery, the fault detection problem is conceptually the hardest. Key to the understanding of the problem is the notion of a fault and the ability of a test to find its occurrence. In off-line system testing, some test pattern is applied to the components of the system. The expected behavior is known, and any deviation from this is flagged as a fault. Systems designed for this type of test fall under the study of *Design for Testability* [McCl85]. The internal circuitry is designed such that individual components can be tested and failed ones located through application of test patterns. Traditionally this kind of analysis has concentrated on low level gate issues.

The problems with off-line testing for faults are twofold. A component that exhibits transient failures may pass an instance of a test. Component exercising attempts to catch these errors by repeatedly testing components during periods of system idle time, however, this off-line testing may not catch transient errors that occur during the actual operation of the system. The second problem is that a test may not be complete. A complex system may have a sufficiently large number of states that generation of a complete test may be infeasible or impossible.

Huang and Abraham in [HuAb84, Huan83] proposed the idea of *Algorithm Based Fault Tolerance* in which checksum encodings are embedded in the data. This is of dubious importance as a fault detection technique as most computing systems already contain some sort of checksumming and error correction at the bit level. One paper [HuAb84], however, proposed the idea of using properties of the solution as acceptance tests. This method is deemed elegant by the following observation:

> *Since we test the intermediate results for correctness with respect to the algorithm, the end solution is correct if the intermediate results*

are correct. If processor errors occur that do not affect the solution, then they are not errors.

For application oriented reliability, this method is clearly superior to component exercising or other off-line tests. The test set consists of only what is necessary to ensure correctness for the current application. The work in application oriented fault tolerance of this type comprises only a small portion of Huang's thesis [Huan83] and the detection method proposed is incomplete. However, the concept is excellent.

1.7 EXECUTABLE ASSERTIONS

The *Assertion* is the basic unit of program verification [Somm82], software fault-tolerance [Rand75], and the application oriented fault-tolerance paradigm described in this thesis. Development of these assertions is straightforward for the sequential program execution environment (See [YaCh75, Andr79, Somm82, HuAb87] for examples). Assertions are primarily generated from the coding and design process.

An assertion is a statement of the form:

If not ASSERTION **then** ERROR;

where ASSERTION is a Boolean invariant on the program state. Assertions integrated into the program code are called *Executable Assertions*. If an assertion fails due to some abnormality, whether hardware or software, the ERROR condition is raised and appropriate action taken. [Stuc77] discusses the use of executable assertions for software fault tolerance, but there is nothing to limit their use in hardware testing.

Assertions are typically outgrowths of the design process as in the Recovery Block Model of Randall [Rand75]. Design generated assertions are desirable from the aspect that they may often be generated automatically from the design language [HuAb87].

In the sequential programming environment, each result of a statement is deterministically a function of the current program state and the statement executed. Procedures and function calls may be handled in one of two ways. Either the function may be considered as a "meta-statement" which is a function of the calling program's state, or the

statements internal to the called function may be expanded inline to the calling program. In any case, assertions are made using the state of a single program and its statements. Careful choice of these assertions, particularly those involving loop invariance, lead to a high degree of confidence in both the program's termination and correctness.

1.8 DIRECTION OF THIS THESIS

Large scale multiprocessors are a product of this decade. Little if any consideration has been given to the system aspect of reliability. As these systems come into widespread usage and as their size and resulting complexity continues to grow, reliability will continue to be a major issue and a viable research topic. This research represents the first such attempt at specifying a reliable parallel processing environment for large scale multiprocessors. This envionment is characterized by the *Reliable Parallel Processing Model* of Figure 1.4.

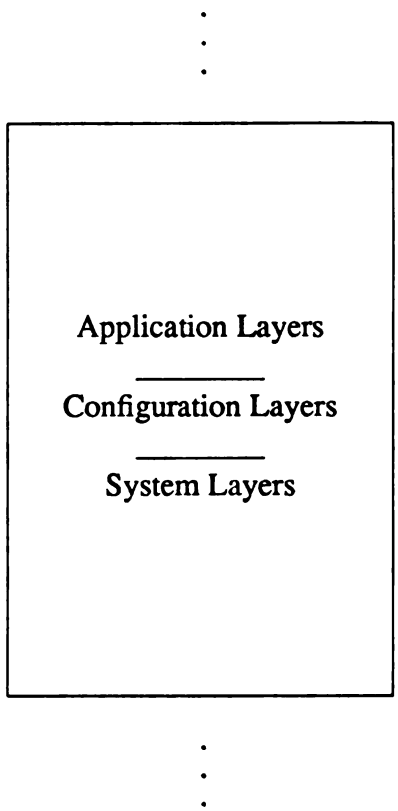

Figure 1.4. The Reliable Parallel Processing Model

The applications programmer need only specify an abstraction of the necessary reliability information at the upper levels. The system environment consists of tools at the lower levels of the model that minimize the burden of reliability programming on the applications programmer. The configuration layers enable the reliable parallel processing model to be employed in various system topologies.

1.9 Thesis Outline

This introduction has provided a historical basis of reliability and addressed some of this issues that are considered in the following chapters.

Chapter 2 gives a motivation for study of new techniques for fault-tolerance. It describes a probabalistic model of expected run time in a large system in the presence of failures as a motivation to provide hardware fault-tolerance. The need for software fault

tolerance, in the form of executable assertions for the parallel environment, is perhaps even stronger motivation for this study than hardware fault-tolerance.

Chapter 3 discusses the fault detection problem, temporally the first problem encountered and perhaps the most difficult to solve.

Chapter 4 presents a formal classification of the basis metrics which can be applied to the problem at hand to generate the necessary executable assertions which compose a structure called a constraint predicate. The usage of the constraint predicate is detailed both as a hardware fault-detection tool and as a software verifcation/fault-detection tool.

Chapter 5 presents the underlying distributed diagnostic basis and it's ideal virtual machine interface to the constraint predicate. Techniques considered are *Vector Byzantine Agreement*, a new form of Byzantine Agreement, and *Distributed Syndrome Testing*.

Chapter 6 presents a set of tools and techniques for analysis of both the coverage and run time performance for a generated constraint predicate.

Chapters 7 through 9 present three examples of constraint predicate development. The first is a common numerical technique for solving very large systems of equations on a parallel processor. The second is a reliable parallelized form of a non-numerical algorithm for computer vision. The third is a reliable sorting procedure. These chapters also deal with programming experiences and implementation details of the algorithms.

Chapter 10 presents directions for future research. These include the outline of a reconfiguration/mapping control module which isolates the application level from topological changes. Directions for localized reliable distributed reconfiguration and recovery are presented.

Chapter 2

Motivation and Problem Statement

The application oriented fault-tolerance paradigm is a reliability method applicable both to tolerance of hardware faults and software faults. The motivational forces behind consideration of each is different.

2.1 HARDWARE FAULT-TOLERANCE

A system is said to be hardware fault-tolerant if, under a bounded number of faulty components, the system continues to satisfy its operational requirements. This bound is obtained through examination of the reliability of the components of the system.

Periodic testing of the system will require additional time to perform the various bounds checking and message interchanges associated with the test. For small problems, this overhead may not be necessary. If the system has a high reliability and the problem incurs no more than a short execution time, the probability of an error occurring during the execution may be very small. For small problems, it may be best to "take your chances" and run the job with no fault-tolerance. However, long run times with large numbers of processors may not be able to finish before an error occurs. Indeed the operational requirements of the GF11 project at IBM [Agar88] specify the execution of algorithms that can run as long as a year on a 500+ processor machine! The expected time between errors is called the *Mean Time To Failure* (MTTF). If the problem run time is long with respect to the MTTF, then reliability techniques are indicated. If the problem run time is short with respect to the MTTF (as in the "small" problem above), reliability

techniques may not need to be used. Furthermore, when an error occurs, the problem must be restarted from the beginning. If the MTTF is short, the job may never complete.

Reliability modeling can provide a basis for quantification of these concepts. We must make some assumptions about the type of problem and solution being attempted.

Model Assumptions	
Serial Reliability	Nearest Neighbor Iterative Problem - All processors must function
i.i.d Exponential	Processor failures are independent, identical exponentially distributed with parameter μ

2.1.1 Failure Model

As noted in the assumptions, processor MTTFs are given by the random variable X and are i.i.d. exponential with parameter μ. The probability density function of X is given by:

$$f(t|\mu) = \frac{1}{\mu} e^{\frac{-t}{\mu}}, \quad t>0 \tag{2.1}$$

The conditional probability that an individual processor fails at some time later than T is given by:

$$1-F(T|\mu) = \int_{T}^{\infty} f(t|\mu)dt = e^{\frac{-T}{\mu}} \tag{2.2}$$

The system MTTF is given by the random variables $X_1, X_2, \cdots, X_N$ where N is the total number of processors involved in the problem solution. Since we have serial reliability, the system fails when one component fails. This is given by the random variable Y:

$$Y = min\{X_1, X_2, \cdots, X_N\} \tag{2.3}$$

Let the conditional cumulative distribution function of Y be $G(t|\mu)$. Since the X_i's are independent,

$$1-G(t|\mu) = \prod_{i=1}^{N}(1-F(t|\mu)) = e^{\frac{-N}{\mu}t} \tag{2.4}$$

Thus $1-G(t \mid \mu)$ is an exponential with parameter $\frac{\mu}{N}$.

2.1.2 Expected Run Time

The expected run time of a non-fault tolerant system is constructed as follows. If the system does not fail before time T, then the job runs in time T. If the system fails before T, then the job must be restarted and the expected run time is the time spent before the failure plus another expected run time (since the exponential distribution has the *memorylessness* property). The expected run time $E(R)$ is given by

$$E(R)=TP_r[Y>T]+E(R)P_r[Y<T]+E(Z \mid Y<T), \tag{2.5}$$

where E(Z|Y<T) is the conditional MTTF given that the system has failed before T time units. Z is given by $P_r[Y=t \mid Y<T]$:

$$P_r[Y=t \mid Y<T] = \frac{P_r[Y=t, Y<T]}{P_r[Y<T]} = \frac{P_r[Y=t]}{P_r[Y<T]} = \frac{g(t)}{G(T)}, \quad 0<t<T \tag{2.6}$$

Since $g(t)$ is given by $\frac{N}{\mu} e^{\frac{-N}{\mu}t}$, we have

$$E(Z)=\int_0^T \frac{N}{\mu} t \frac{e^{\frac{-N}{\mu}t}}{1-e^{\frac{-N}{\mu}T}} dt = \frac{\mu}{N} - \frac{T}{e^{\frac{N}{\mu}T}-1}. \tag{2.7}$$

Solving (2.5) for $E(R)$ and using the value for $E(Z)$ obtained in (2.7) we obtain

$$E(R)=T - E(Z) + E(Z)e^{\frac{N}{\mu}T}. \tag{2.8}$$

2.1.3 Discussion

The $E(Z)$ is actually optimistic since it is assumed that when a processor fails, the system halts. In Chapter 5 it is shown that this is not always the case for Byzantine faults. If it cannot be determined that a processor has failed until the end of the computation, then $E(Z)$ is replaced by the larger value of T and $E(R)=Te^{\frac{N}{\mu}T}$. The general shape of (2.8) is given in Figure 2.1 for fixed values of N = 32768 and μ = 5000 hours.

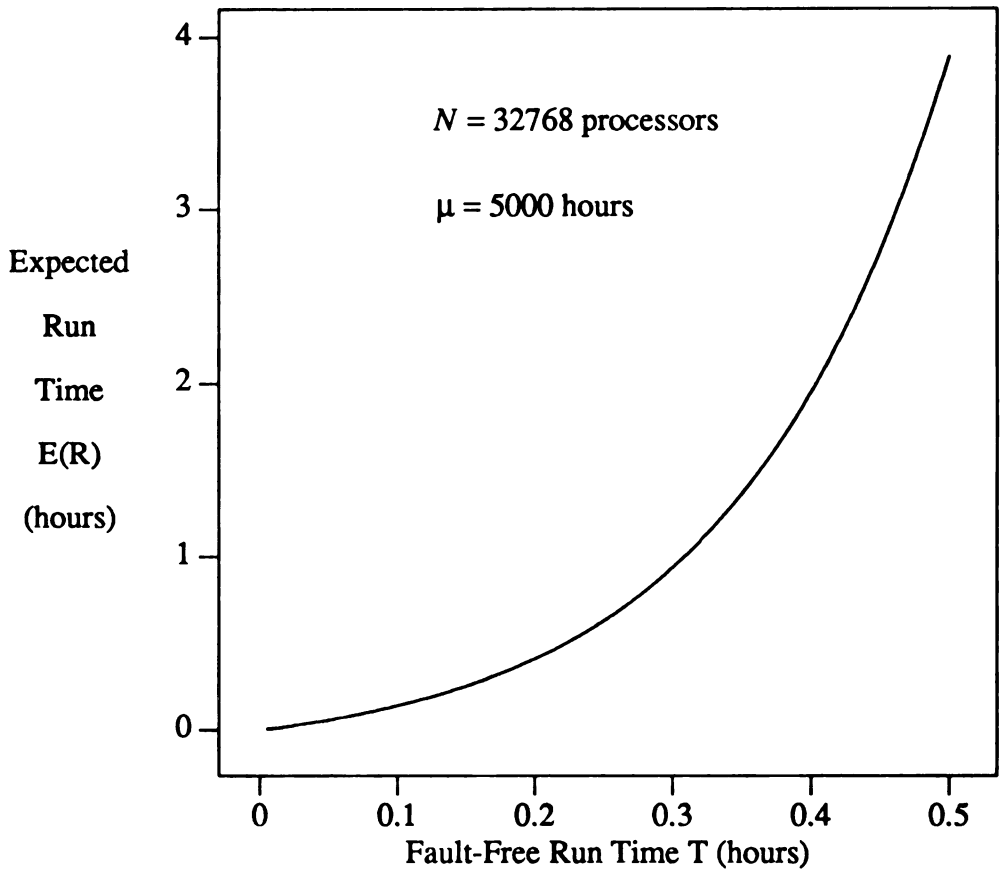

Figure 2.1. Expected run time for a serial-reliability parallel algorithm

The abscissa shows the time for a solution with no fault tolerance that encounters no errors. The ordinate is the expected run time given the reliability of an individual processor and the number of processors involved in the solution. Since this is a serial reliability model, a single processor failure causes a job restart.

2.2 SOFTWARE FAULT-TOLERANCE

A system is software fault-tolerant if, under some bounded number of software design/coding errors, the system continues to satisfy its operation requirements. Notice that this is closely related to the notion of hardware fault-tolerance discussed in the

previous section. However, this is where the similarity ends.

Software faults result from design and coding errors. These errors are much more prevalent than errors that one encounters from hardware design. Furthermore, verification of hardware designs, while a non-trivial task, is still a manageable problem. The large number of program states for even a trivial program makes verification of general software nearly impossible. Testing can only show the presence of errors, not their absence. Operational fault-detection, in which executable assertions become part of the operational environment, can detect faults during the system's operational lifetime. Operational fault tolerance may be achieved by Randall's Recovery Blocks [Rand75] in which alternate algorithms are employed if a software failure occurs. As noted in the introduction, this system is in limited use in the Space Shuttle flight computers.

The DMMP environment limits the amount of information available to an executable assertion. Since a checking processor cannot examine another's internal memory, the message sent is the only form of communicated information available.

To develop and implement executable assertions on a DMMP requires additional consideration over the sequential environment. Since processors may only communicate via messages, detailed assertions may not be implementable. In general, the detailed assertions do not occur at *testable stages*, i.e. message interchange points. Thus, the entire program state is no longer deterministically obtainable by any single processor. These considerations require development of executable assertions that function in the presence of partial information.

These DMMP issues foster consideration of granularity of test design and specification as a primary issue. Creation of assertions at the message interchange granularity is essential if true distributed diagnosis of hardware faults is to be achieved. Fortunately, as a problem is decomposed into its parallel components and partitioned/mapped onto the parallel processor, the message exchanges fall at the problem specific parallel component boundaries. Assertions then are a function of the local

program state and any received messages from other parallel components.

The primary question is how to generate, in a systematic way, assertions at this level. This is the major topic of this thesis. The metrics necessary to guide this generation are described in the Chapter 4.

2.3 PROBLEM STATEMENT

The attainment of reliable parallel processing can proceed along any of the paths given in Figure 1.1 (p. 7). The most promising method is that of reliable fault detection, reconfiguration, and recovery [Garc82, YaHa84, Stro86]. Solution of all three problems is a massive task. The fault detection problem is temporally the first to be considered. As mentioned earlier, fault detection is the most difficult problem due to the elusive nature of the definition of a fault. Fault detection can be considered as a twofold problem. The first is to develop the tools necessary for consistent diagnosis in the parallel environment. Using this consistent diagnosis, the second problem is to develop classes of executable assertions that can be reliably applied to detect faults. In Figure 1.4 (p. 17), the consistent diagnosis resides in the system layers, and the executable assertions reside in the application layers.

The approach taken in this research is software oriented. This provides the most technologically independent result and the most complete fault coverage possible. The major contribution of this dissertation is the development of tools for consistent diagnosis in the Byzantine environment and specification of metrics for development of constraint predicates for a wide range of problems. The result is an integrated technologically independent framework of reliable parallel processing for large multiprocessor systems.

2.4 CHAPTER SUMMARY

This chapter has presented a motivation for the study of reliable parallel processing both from a hardware and software reliability standpoint. Large multiprocessing systems with thousands of components will experience low overall system reliability. Thus, system-level techniques must be employed to achieve reliable system operation. The problem of providing reliable software is not as easy to quantify, but its need is easy to see. The detection (and recovery from) design/coding faults is at the heart of software fault-tolerance. To apply software fault-tolerance techniques to a DMMP requires a different approach. This is detailed in the next few chapters.

Chapter 3

A New View of the Fault Detection Problem

Analysis of the fault detection problem is interesting as it is temporally the first and perhaps most difficult problem encountered. To examine techniques for fault detection in the applications environment, fault detection must be viewed in a new way, one that minimizes the impact of system issues on the applications programmer. This chapter explores the concepts of *distributed fault diagnosis*. A new model for fault detection is presented and a system-oriented classification scheme given for the "size" of the tests employed.

3.1 THE DISTRIBUTED FAULT DETECTION MODEL

The goal of distributed fault diagnosis is to achieve a diagnosis in which all processors involved in the diagnosis (the *diagnosticians*) come to the same conclusion - one that is representative of the true state of the system - in the presence of faulty behavior under a bounded number of faults. To state the problem more succinctly, the following distinction is made. Using standard terminology, a distributed *Correct* diagnosis is one in which all diagnosticians arrive at the same decision on the set of faulty units, and all units flagged as faulty are indeed faulty. A *Complete* diagnosis is one in which all faulty units are detected. Thus, optimally, distributed diagnosis achieves a correct, complete diagnosis among the diagnosticians. In a DMMP, the diagnosticians are the peers of the tested processor although, in general, the diagnosticians may also be specialized diagnostic units. The units that receive the results of the test then take an appropriate action such

as halting the system or initiating reconfiguration.

The fault model considered (some of which are listed in Table 1.1 (p. 10) has a direct impact on the correctness and completeness of a fault-detection algorithm's diagnosis. In the unrestricted fault case, messages may be lost, delayed, or altered by a faulty resource. Such behavior is called *Byzantine* behavior [LaSP82]. Furthermore, a Byzantine faulty processor may send different versions of the same message to different processors. This makes correct operation in the presence of these faults difficult not only to mask but to detect.

A more restrictive fault case is that proposed for testing by [PrMC67]. In this model (the PMC model), a faulty processor, when queried for its status by a non-faulty unit, always responds that it is faulty.

Fault diagnosis under the PMC model (and more restrictive models) has been studied extensively over the past two decades. Between the case of the PMC model and the Byzantine model, [MaMa78, YaMa86a] have studied the class of "intermittent faults." A result of a test for faulty behavior that is correct for the PMC model may be incorrect for the intermittently faulty behavior model. The Byzantine model is the same basic fault model as intermittently faulty behavior but with the additional stipulation that failure is not random, but malicious. The distinction between intermittent faults and Byzantine faults is subtle, but important. Fault diagnosis under the Byzantine model has been studied recently by [ShRa87] as true Byzantine diagnosis considering faulty message relays as faulty behavior. Chapter 5 considers, instead, faulty message content as faulty behavior. Correctness and completeness of the diagnosis uncover important differences between these two approaches.

For distributed diagnosis to achieve its goal, diagnostic algorithm design must consider not only the difficulty of faulty resource behavior in the protocol or message interchange necessary to communicate the diagnostic information, but must also include an appropriate test for faulty behavior. In the Byzantine environment, for example, it is not

always clear what a test would be. As for intermittent faults, a faulty processor is perfectly free to pass an instance of a test and yet fail during the actual system operation. Thus test development is at least as hard as development of the diagnostic algorithm itself.

To study fault detection we split the distributed diagnosis problem into two parts as shown in Figure 3.1, thus, formalizing the terminology as follows.

Acceptance Test The goal of the acceptance test is to detect erroneous system behavior.

Diagnostic Basis The goal of the diagnostic basis is to provide a reliable environment, in the presence of faults, such that the information delivered to the acceptance test can result in a correct and complete diagnosis.

This splitting allows development of the testing basis to proceed independently of actual test development. Proof of correctness then, can similarly be separated. Proof of assertions of diagnostic consistency and correctness can be made abstractly in the general distributed system sense. Proof of the acceptance test is based solely on program-specific assertions and follows the model of more traditional software engineering approaches. An excellent survey of these techniques is provided in [AdBC82].

A further dichotomy of these two classes of algorithms exists in the orientation of their specification. The diagnostic basis can be considered as a "system oriented" concept in the form of a set of tools for programming support. The acceptance test can be considered as an application level, "programmer oriented" concept. This dichotomy, ideally, is complete. However, system issues may pervade to the application level resulting in a higher degree of coupling between these layers than is desired.

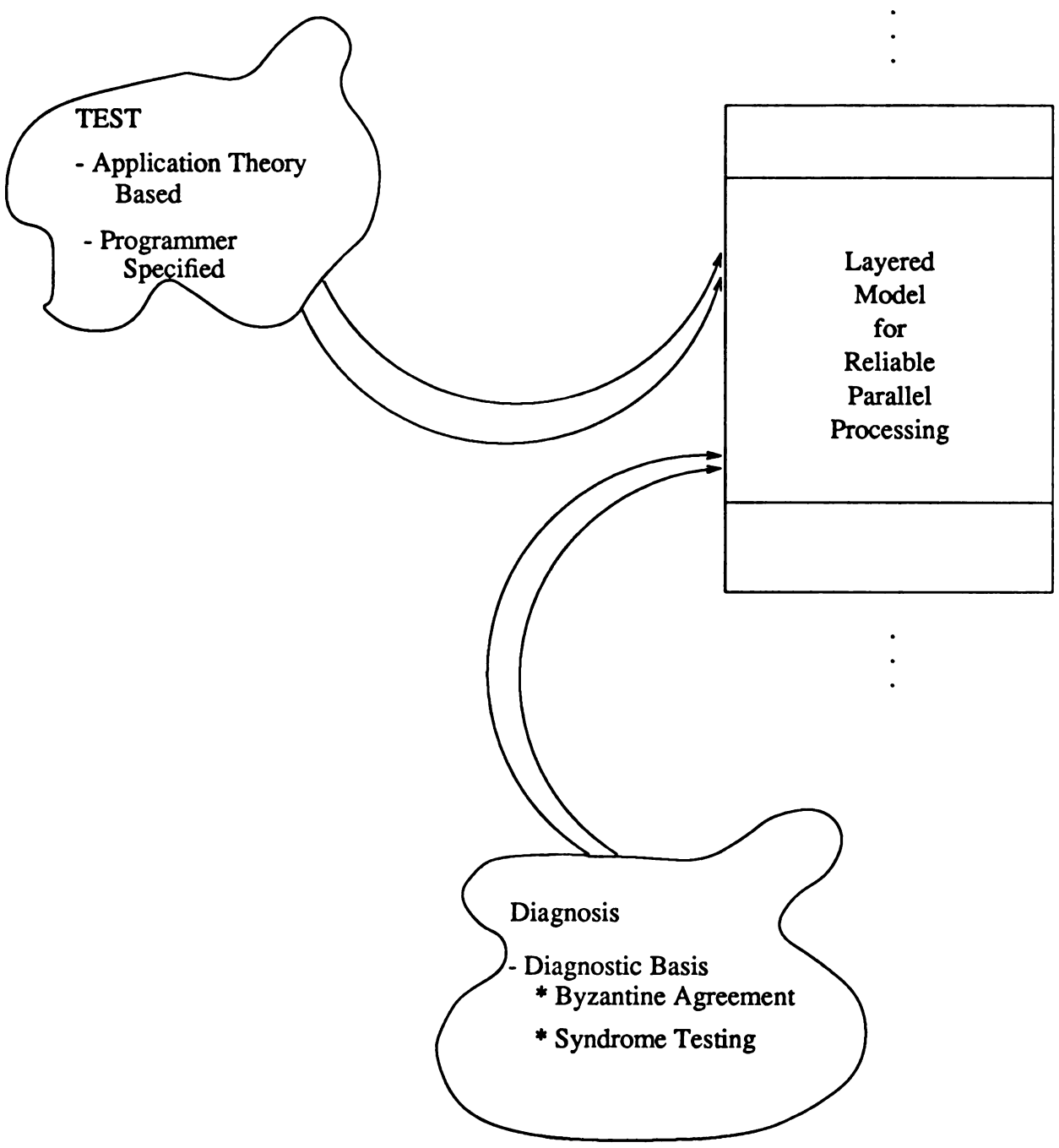

Figure 3.1. Splitting the Detection Problem

3.1.1 Diagnostic Basis

The job of the diagnostic basis is to provide the acceptance test layer with sufficient information to reach a correct, complete diagnosis. In the distributed environment, this

requires the diagnostic basis to provide the same local information to each diagnostician. The diagnostic basis must preserve this information such that a non-faulty processor cannot be erroneously flagged as faulty, nor must it allow a faulty processor to escape detection.

In the literature, two techniques have formed the mainstream of distributed diagnosis. The first are ad hoc approaches that work with limited fault sets. Ad hoc approaches are not attractive since they have no formal theoretical design basis and thus do not lend themselves to analysis. Nothing further will be said about these techniques here.

The second, more comprehensive testing scheme, is called *syndrome testing*, first proposed by [PrMC67] for the PMC model of faulty behavior. In this approach, each peer tests some of its neighbors and, in turn, is tested by other neighbors. The set of testing assignments is called a testing digraph (see [KuRe86] for a survey of testing techniques). The test results form the *syndrome* of the system. For classes of well defined testing digraphs, the syndrome provides the diagnosis algorithm with sufficient information to yield the set of faulty processors.

Syndrome testing in the Byzantine environment (for deterministic models) is incomplete (Chapter 5). This incompleteness is inconsistent with the stated function of the diagnostic basis. A further limitation is that syndrome tests do not integrate well with the concept of a fault-tolerant algorithm. The primary advantage of the fault-tolerant algorithm is that the exchanged messages form both the actual algorithm communication and the test. Syndrome testing can be modified to utilize the actual data values as testing weights. The main problem in the application environment is that different processors may accept different versions of the same message due to a faulty processor. If the actual application is testing, then this is not a problem as the application data are simply the testing weights - binary values *good* or *bad*. This view of the system test is indeed consistent with our notion of application oriented tests for the specific application of

testing. For general applications, however, correct diagnosis may still lead to incorrect operation.

These problems lead to consideration of a third, newer, type of testing called *Byzantine Detection*. Solution of the Byzantine detection problem utilizes *Byzantine Agreement* [LaSP82]. In Byzantine Agreement, a group of processors, each having some local data which they wish to reliably communicate to other members of the group, engage in an agreement algorithm. At termination of the algorithm, each processor holds its view of values held by the other processors subject to the following conditions: (1) Any two non-faulty processors in the agreement obtain common values for all processors in the agreement, and (2) if a processor k is non-faulty and wishes to communicate a value to the other members of the diagnosis, then at the end of the agreement, each other non-faulty processor receives k's intended value. An algorithm which allows the non-faulty processors to achieve the above conditions solves the *Byzantine Generals Problem.* There are many versions of Byzantine Agreement which work under differing assumptions of synchronization, connectivity, and probabilistic correctness. A detailed description of Byzantine Agreement is presented in Chapter 5.

The Byzantine Generals problem has been the subject of study as a solution to the problem of finding a consensus among a number of processors in a faulty environment in which the total number of processors, n, contains at most t faulty processors. The Byzantine algorithms can be thought of as distributed voting algorithms which mask faults. While Byzantine Generals solution algorithms mask faults, alone they have a drawback in the solution of the fault-detection problem. Examples can be constructed (and indeed this is the method used by [ShRa87]) in which it is obvious to all members of the agreement group which processor is faulty. Byzantine Agreement does no detection; it only guarantees that a consistent state of the system is produced for each processor. This, of course, solves one of the stated distributed system problems. However, for reasons that are explored in detail in Chapter 5, to provide the necessary fault coverage, we must

appeal to the acceptance test. Thus, masking of faulty behavior by the diagnostic basis through Byzantine agreement does not compromise complete diagnosis as it does in syndrome testing. Note that unlike syndrome testing, each processor has not just the same diagnosis, but has the same set of values to operate on. This is key to the Byzantine Detection problem.

3.1.2 Acceptance Test

Assume that a diagnostic basis is employed as a reliable message delivery system for message exchange between peers. In a DMMP, since the only method of communication is message passing, it would seem that a faulty processor can remain undetected as long as it continues to send messages. However, faulty processors can be detected through examination of their messages in the context of the system as outlined in the beginning of this section. The construct that forms the basis for the acceptance test is the *Constraint Predicate*. The constraint predicate is comprised of executable assertions which form a functionally cohesive unit (in the software engineering sense). The actual development of the constraint predicate is deferred to Chapter 4.

The constraint predicate must satisfy the following property. Any two processors that are correct, given the same input values, reach the same conclusion. Thus any neighboring processors working on the same portion of the problem can execute the same constraint predicate and reach a common decision regarding their immediate neighbors, and thus report the same conclusion of error or no error. Furthermore, since each correct processor has obtained and verified the same data from each processor under consideration, the solution of the problem can proceed correctly in the presence of any errors that are maskable by the diagnostic basis.

Fault model considerations at this level are different than that at the diagnostic basis level. The Byzantine model, while appropriate to consider for the diagnostic basis, is unnecessary for the acceptance test. Hardware errors that result in intermittent faults

(real physical phenomena) will manifest themselves as Byzantine failures. A failure in the diagnostic basis can (and will in the Byzantine environment) cause such pathological errors ranging from inconsistent diagnosis to deadlock. It is the function of the reliable distributed diagnostic basis to intercept, filter, and mask errors. For this task, it must be tolerant of Byzantine faults. In contrast, errors in the actual application data can cause, at worst, increased *fault latency* - the time to locate a fault once it has occured. The reliability required at this level is less stringent and lends itself to test development with probabilistic coverage of faults as the main goal.

We now turn our attention to the specific tools afforded the applications programmer as a result of our development so far. Consideration of the programmer's point of view dictates ease of specification of the constraint predicate. This is fostered purely by the human factors involved in programming. If something is difficult to specify, then it won't be utilized. The better the tools are, the easier application oriented constraint predicate generation will be.

There are a set of fundamental objects that a parallel application must utilize: a data structure that holds communicated information and message communication primitives. Schemes similar to the Remote Procedure Call (RPC) mechanism proposed by [BiNe84] or the unified distributed computing environment proposed by [Gend87], both of which provide a sufficiently high abstraction for the programmer, can be utilized as message communication primitives.

3.2 PROGRAMMER/SYSTEM ISSUES

The constraint predicate provides the application programmer with the ability to control the actual test while minimizing the concern for system issues. However, system issues such as performance and topology for fault-tolerance still must be considered. To analyze a constraint predicate's performance, an interface oriented classification based on the cardinality of the test size is proposed. This presupposes that there is a direct

relationship between test size and data availability.

How much input is needed for a test? Empirically for any test, we want the number of processors that must be queried for input to be as small as possible. Table 3.1 summarizes a predicate classification that will be useful in constraint predicate development for an N processor system.

Predicate Class	Predicate Name	Size
IPT	Individual Predicate Testable	1
GPT	Group Predicate Testable	$1<$ and $<<N$
PPT	Problemwide Predicate Testable	N

Table 3.1. Constraint Predicate Size Classification

IPT is the most desirable in terms of run time cost. It also maps well into existing fault-detection schemes such as syndrome testing in which a single processor tests another single processor. IPT may not be sufficient for many problems, however, as not enough information for a good diagnosis may be available. Many IPT tests are satisfied locally, but the global result is incorrect.

GPT utilizes several values reported from processors to generate a correct diagnosis. Many DMMP applications are of the nearest neighbor variety. Thus the data values that are relayed by processors in close proximity are related to the testing processor. The more values that the test uses, the better the test. When the test uses all values of the neighbors of the testee, then the tester has enough information to completely replicate the testee's calculation. Thus GPT predicates can be used to implement N-Modular Redundancy.

PPT requires all values of a problem to be checked. Clearly in a large scale DMMP, this is infeasible except if the test is done very infrequently. Each value obtained incurs extra communication overhead, an expensive commodity in a DMMP.

The predicate classes above give the notion of the size of the test. Also important is the fault-latency. Optimally a test finds an error as soon as it occurs or with a fault-latency of one, the one standing for one cycle or message exchange in the tested system. Other tests, in reality may have much longer fault latencies. Typically as we move from IPT to GPT to PPT, the fault latency becomes shorter. Thus the test class used is selected as a tradeoff between time complexity and tolerable fault latency.

3.3 CHAPTER SUMMARY

The new view of fault-detection isolates most of the "system-oriented" issues from the applications programmer. These system oriented routines include reliable broadcast and detection support. The level of coupling between these two layers is low. The applications programmer then must specify only an abstraction of fault-tolerance that is based on the application at hand. The next chapter details development of this abstraction for the distributed parallel envrionment. Chapter 5 describes considerations and possible algorithms for the distributed diagnostic basis.

Chapter 4

Constraint Predicates - A Programmer Level View

The development of the necessary constraint predicate is at the crux of application oriented reliability. Historically, the main shortcoming of application oriented reliability has been the difficulty of finding, and of how to find, such a constraint predicate.

4.1 Definition

Definition 4-1: A constraint predicate Φ
1) constrains processor errors to within acceptable (testable) limits,
2) attains its functionality from the natural constraints of the application problem, and
3) never erroneously flags a correct result as faulty.

Condition (1) is, intuitively, a test. Each testable portion of a calculation must conform to the expected result. Given the granularity of assertion to be at the processor level, a tester investigates messages from the external environment for erroneous content and format. Assuming that we have employed a proper testing basis as our tool for fault tolerance, it can be assured that the designated testers will also make the same diagnosis, and thus any seemingly faulty processor will be forced to remain within correct limits by its neighboring processors. Any deviation from these limits will be flagged as a fault and the appropriate action may be taken. What remains to be understood is how to find appropriate constraint predicates.

Condition (2) is best introduced by contrasting it with traditional fault tolerance techniques such as replication through N-Modular Redundancy. For replicated calculations, regardless of how the result is obtained, an error is flagged if the replicated results

differ. Constraint predicates make use of known characteristics of the problem. A very simple example of a constraint predicate is to consider some predicate Φ_{F^A} (the meaning of Φ_F will be made clear in the next few paragraphs) for the calculation of the area of a plane figure.

Predicate Φ_{F^A} (area_result:**real**) **returns bool**;
    if (area_result < 0)
      ERROR;
end.

Clearly any area result that was negative would be in error. Thus Φ_{F^A} is correct in the sense that it does not erroneously flag a correct result as in error. However, it has woefully poor error coverage and is therefore incomplete. The faulty processor would be free to choose any positive real number for "area-result." Thus one should intuitively feel that the constraint predicate Φ_{F^A} for area should be considered unacceptable. Exactly why this situation is present will be revealed in the following discussion of constraint predicate generation and coverage analysis.

A more effective constraint predicate is illustrated by considering the problem of sorting a list of elements. Formally, sorting is defined as follows:

Definition 4-2: Given an input list $\mathbf{I}=(I_i), i=0, \cdots, N-1$, a sorting procedure S finds a permutation $\Pi=(\pi_i)$ such that:

1) $$I_{\pi_i} \le I_{\pi_{i+1}}, i=0, \cdots, N-2$$

or

2) $$I_{\pi_i} \ge I_{\pi_{i+1}}, i=0, \cdots, N-2$$

Let the output list delivered by S be $\mathbf{O}=(O_i)=I_{\pi_i}$ and assume that an ascending sort is performed. The following theorem immediately follows.

Theorem 4-1:

1) If $\exists i$ such that $\forall j$, $O_i \neq I_j$ or $\exists i$ such that $\forall j, I_i \neq O_j$,

or

2) $O_j > O_{j+1}$ for some j,

then the result **O** produced by S is incorrect.

Proof: For part (1) if either there is no j such that $O_i=I_j$ for some i or there is no i such that $I_i=O_j$ for some j, then the output list **O** is not a permutation of **I** thus violating Definition 4-2. For part (2), the output list must appear in a non-decreasing manner as an ascending sort it being performed. □

From Theorem 4-1, the constraint predicate Φ_{F^S} follows immediately.

```
Predicate Φ_F^S (I,O,N) returns bool;
    for i := 0 to N−1
        if O_i≠I_j for every j then
            ERROR;
    for i := 0 to N−1
        if I_i≠O_j for every j then
            ERROR;
    for j := 0 to N−2
        if O_j>O_j+1
            ERROR;
end.
```

Φ_{F^S} is correct since it flags any unsorted output list as an error, and it is complete since any output list that is not a permutation of the input list constitutes an error. However, this predicate can only be applied at the termination of the sorting calculation. If an error occurs early in the calculation, this predicate cannot find it. Thus the fault-latency is long. We shall re-visit sorting predicates in Chapter 9.

Condition (3) is necessary to prevent unnecessary loss of computing resources. An individual component of a constraint predicate might perform an incomplete subtest - that is the subtest allows faulty behavior to pass. It will never, however, perform an incorrect test in which a correct result is flagged as faulty. At some stage of testing, the subtests will either progress sufficiently to actually flag the error, or the actual error will be caught by another test that comprises the constraint predicate. The intersection of all

such tests, each of which is correct, composes a correct constraint predicate.

Two issues are important in constraint predicate development for the distributed parallel environment. Granularity, as mentioned in Chapter 2, is the first. A more general issue common to executable assertions is *coverage*. We shall treat the granularity issue now and delay the issue of coverage until Chapter 6.

Problem mapping to the DMMP environment is usually done in the *Data Parallel* [HiSt86] style. Each processor executes the same portion of an algorithm on different data. For maximum parallelism to be obtained from this type of mapping, the problem must contain a natural structure that allows the parallelism to be achieved. Furthermore, the identifiable points of parallelism will naturally fall at message exchange boundaries for data parallel algorithms. Thus the constraint predicate is more closely allied with the problem than with any specific solution. This relationship is shown in Figure 4.1 which depicts the construction of the constraint predicate.

4.2 BASIS METRICS

The following is proposed as a basis for constraint predicate generation through property extraction. In a solution, each testable intermediate result should satisfy one or more of the three predicate subclasses of progress (Φ_P), feasibility (Φ_F), and consistency (Φ_C).

4.2.1 Progress (Φ_P)

We require progress to be made at each testable step of the solution. Progress means that the state of the solution advances to the goal or final solution of the problem. Each testable step of the solution is defined as a message interchange, sub-message interchange, or multi-message interchange dependent upon the type of solution. If this progress is not made, then faulty behavior may indefinitely postpone the solution - any solution, even an incorrect solution.

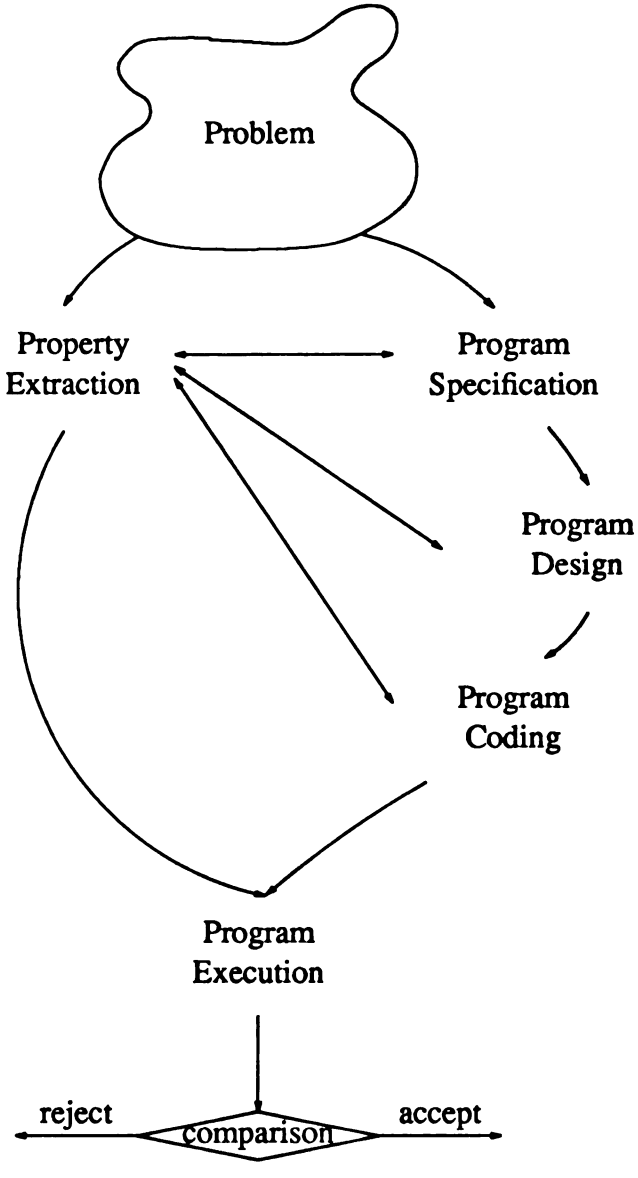

Figure 4.1. Abstraction of the

Constraint Predicate from the Software Life Cycle

A partial dichotomy exists in parallel algorithms. Problems in which the number of steps is known *a priori* forms one class, and problems of an iterative convergent nature in which the number of steps is not known *a priori* forms the other.

Clearly for the class of problems with a known number of steps, each processor must complete the required number of steps. Early termination constitutes an error. Let the current step be k. If we bound the time interval that a processor can take to produce a step of the problem (*timeout*), then the following predicate can be used to check for progress of P_j (Note the use of the Ada **select** construct which nondeterministically chooses between any true alternatives if the choice is not unique).

```
Predicate Φ_P (max_step,timeout,k) returns bool;
    for i = 1 to max_step do
        select
            read data from P_j;
            if data not from the k'th step
                ERROR;
        or
            delay timeout;
            ERROR;
        end;
        process locally;
end.
```

Note that if a processor continues to send messages after *max_step*, this is not considered an error for this behavior does not affect the final result of the calculation.

Common to both iterative and non-iterative problems is the notion of convergence. We shall assume that the problems attempted are convergent; for if not, then a nonconvergent result is indistinguishable from a hardware failure. Most, if not all, problem solutions contain implicit information necessary to form a *convergence envelope*. This places bounds on the reduction of error between the current and final solution. Let the error at step k of the solution be

$$\varepsilon_i^{(k)} = |u_i^{(k)} - u_i|, \text{ for } i \in \{1,...,N\}$$

where k is the current step count of the solution, $\mathbf{u}^{(k)} = (u_i^{(k)})$ is a single (or vector-valued) state of the solution at the k'th step, and $\mathbf{u} = (u_i)$ for $i \in \{1, \cdots, N\}$ is the actual

correct solution. Let $\|\cdot\|$ be some suitably chosen vector norm, scalar absolute value, or discrete convergence function†. Let $\boldsymbol{\varepsilon}^{(k)}=(\varepsilon_i^{(k)})$, for $i \in \{1, \cdots, N\}$. Monotonic reduction of error is then defined as

$$\|\boldsymbol{\varepsilon}^{(k+1)}\| < \|\boldsymbol{\varepsilon}^{(k)}\|$$

$$\|\mathbf{u}^{(k+1)}-\mathbf{u}\| < \|\mathbf{u}^{(k)}-\mathbf{u}\| \tag{4.1}$$

A suitable constraint predicate Φ_P for this relation is:

```
Predicate ΦP (u(k),u(k+1), real vector-valued, timeout,k) returns bool;
    select
        read u(k+1);
        if (||u(k+1)−u|| ≥ ||u(k)−u||)
            ERROR;
    or
        delay timeout;
        ERROR;
    end;
end.
```

Two things should be said about this predicate. In the programmer/system classification scheme this is a PPT predicate. It requires all the values held by all the processors involved in the calculation. For a mesh of N processors this takes $O(\sqrt{N})$ time, and for a cube of size N this takes $O(log_2 N)$. Both these values are non-negligible. The second difficulty is that it involves the unknown $\mathbf{u}$. What is more desirable is a constraint predicate that can test locally in $O(1)$ time, or even unit time, and only use obtainable data for its tests. Locality and obtainability cooperate to form an IPT test for monotonic convergence.

Consider as a special case of the convergence envelope localized monotonic reduction of error. This is typically seen in relaxation solutions such as PDE relaxation.

† A discrete convergence function is employed in Chapter 9 when parallel sorting is considered.

Without loss of generality, assume that the sequence of local errors $\varepsilon_i^{(k)}$ is monotone decreasing.

$$\varepsilon_i^{(k+1)} \leq \varepsilon_i^{(k)}$$

$$u_i^{(k+1)} - u_i \leq u_i^{(k)} - u_i \tag{4.2}$$

It then follows immediately that the sequence of u_i's are also monotonically decreasing. A suitable constraint predicate Φ_F for this relation is:

Predicate Φ_F $(u_i^{(k)}, u_i^{(k+1)}$, **real**, $i,k)$ **returns bool**;
    If $(u_i^{(k+1)} \geq u_i^{(k)})$
      ERROR;
end.

Convergence envelopes, while ensuring a guarantee of nonegative progress, cannot check for *sufficient progress*. By this we mean that a solution satisfying the above predicate may proceed arbitrarily slowly. Consider *bounds* on the convergence rate $\gamma_{\min}^{(k)}$ and $\gamma_{\max}^{(k)}$ such that each step satisfies

$$\gamma_{\min}^{(k)} \leq \frac{\| \boldsymbol{\varepsilon}^{(k)} \|}{\| \boldsymbol{\varepsilon}^{(0)} \|} \leq \gamma_{\max}^{(k)} \tag{4.3}$$

Again this is a PPT class predicate. A corresponding local rate bound may be obtained and implemented similarly to that of (4.1).

Progress alone is not sufficient to guarantee solution correctness. The final vector, $\mathbf{u}^{(k)}$, particularly when the bounds γ are loose, may differ significantly from the actual solution $\mathbf{u}$. To restrict completely arbitrary behavior further, we consider feasibility as the next constraint predicate basis.

4.2.2 Feasibility (Φ_F)

Each testable result must remain within the defined solution space of the problem. Formally, consider a solution space H_k and any intermediate result $\mathbf{u}^{(k)}$. Then for any

step k,

$$\mathbf{u}^{(k)} \varepsilon H_k$$

The feasibility constraints are often immediately apparent from the nature of the problem studied. Problems in physics and engineering such as equilibrium problems, eigenvalue problems, and to a lesser extent propagation problems contain feasibility constraints in the form of boundary conditions. Indeed these boundary conditions are exactly the class of natural problem constraints. Equilibrium problems can be described as *jury* problems in which the entire solution is obtained by a jury which must satisfy all the boundary constraints and all internal requirements of the problem. Propagation problems can be considered as *marching* problems in which the solution marches from the initial state guided and modified by the side boundary conditions. In these type of problems, the boundary conditions are of course known *a priori* and do not vary as the solution progresses ($H_k = H$ is stationary), and thus can easily be used as feasibility constraints.

Branch and bound tree searching (for example see [HoSa84]) is a method which searches a state space (tree) for a minimum cost goal state. This search is guided by an accumulated and estimated cost function. Let $c^{(k)}(x)$ be the estimate of the cost function at step k and $c(x)$ be the actual cost of reaching the goal state from node x. If we enforce that $c^{(k)}(x) \leq c(x)$ for $k \in \{1,2,...\}$, then $c^{(k)}(x)$ provides a lower bound in H_k on the cost of any solution obtainable from node x. Let L be an upper bound on the cost of a minimum cost solution (initially L may be ∞). If at some step k, a possible goal node y is discovered, then all nodes x with $c(x) \geq c^{(k)}(x) > c(y)$ may be killed, and L is updated as $L = c(y)$. Thus branch and bound provides a way of dynamically narrowing the bounds on the feasible solutions H_k.

For some problems, the feasibility constraints may be so loose as to be virtually of no use. This is precisely the case for the "area" predicate Φ_{F^A}. This is an example of a feasibility predicate for which $H_k = \{x \mid x \geq 0\}$ for $k \in \{1,2,...\}$ i.e. the set of positive

reals. The difficulty lies not in the idea of using a constraint predicate. This is the best predicate that can be obtained from the problem statement. The problem simply contains insufficient natural constraints from which to generate a good predicate.

4.2.3 Consistency (Φ_C)

Many intermediate calculations contain additional properties that are indirectly obtainable from the problem's natural constraints. These are defined as *Consistency Conditions*. In a white box testing environment, all facets of each testable step can be checked. As noted previously, in the DMMP environment, due to the locality of information, black box testing must be utilized. A consistency predicate may be applied only to the received information and locally known information. This may appear to severely restrict the functionality and usefulness of this type of test, but in reality a consistency test is powerful. [AyOz87] developed an entire constraint predicate using only consistency conditions.

Consistency can compensate for limitations in the progress and feasibility bases. Consider a problem in which (4.3) has a globally specified bound γ but no locally specifiable γ_i. Furthermore, assume that it is too expensive to implement a PPT predicate. Knowledge that a processor has about its local state and the values that it sent to other processors in previous steps can provide bounds on the range of acceptable values for the current testable step.

Assume that the solution proceeds as in (4.2). Let each new value of $u_i^{(k)}$ be calculated as a linear function f of the neighboring values $u_l^{(k-1)}$, $l \in R$ with coefficient a_l. From the perspective of a processor P_i calculating u_i, a candidate intermediate result $u_l^{(k)}$, $l \in R$ must satisfy the following property

$$u_l^{(k)} \le u_l^{(k-1)} - a_l(u_i^{(k-2)} - u_i^{(k-1)}), \quad l,i = 1,2,...,N, \; k \in \{2,3,4,...\} \tag{4.4}$$

To prove this as a theorem it is necessary to consider two successive iterations $u_l^{(k)}$ and $u_l^{(k-1)}$. Subtracting the two yields an expression which is a function of known and

unknown values (from processor P_i's point of view). If we let $W^{(k)}$ represent the unknown values at iteration k, the expression becomes

$$u_l^{(k)} - u_l^{(k-1)} = W^{(k)} - u^{(k-1)} + a_l(u_i^{(k-1)} - u_i^{(k-2)}), \quad l,i = 1,2,...,N, \; k \in \{2,3,4,...\}$$

Since f is a monotonic linear function and $u_l^{(k)} \leq u_l^{(k-1)}$ where $l,i = 1,2,...,N$, $k \in \{1,2,...\}$, then $W^{(k-1)} \geq W^{(k)}$. This provides a lower bound on the amount of movement that processor P_i can expect which results in the inequality given in (4.4). The lower bound is solely a function of the values that P_i sent P_l in the previous iterations and P_l's previous iteration. Furthermore, this is an IPT predicate which makes it attractive from a computational aspect.

4.3 CHAPTER SUMMARY

A systematic way of generating executable assertions for the DMMP envrionment is essential to the application oriented fault-tolerance paradigm. The three predicate subclasses of progress, feasibility, and consistency formulate the constraint predicate. Error coverage analysis techniques for the resulting constraint predicate are presented in Chapter 6. Natural problem constraints are a necessary condition for successful constraint predicate generation. It is not known if natural constraints are a sufficient condition. This is discussed in Chapter 10.

Chapter 5

Distributed Diagnostic Basis

In this chapter, the two diagnostic basis algorithms mentioned in Chapter 3 are explored in greater detail. We present syndrome testing and show that in the Byzantine environment it produces an incomplete diagnosis. As an alternative, Byzantine Agreement is described and an implementation called *Vector Byzantine Agreement*, which is necessary for efficiency in the DMMP, is presented, proven correct, and given running time bounds.

5.1 SYNDROME TESTING AND DIAGNOSTIC IMPOSSIBILITIES

In the mid to late sixties, fault diagnosis research had no real formalism to rely upon. In a landmark paper [PrMC67], the authors outlined a method of system level diagnosis called *syndrome testing*. Peer testing using this method has received concentrated research treatment over the ensuing two decades since its inception and thus is worthwhile to consider as a candidate distributed diagnostic basis.

Syndrome testing of a system S is modeled as a testing digraph $G(V,A)$, where each processor in S is represented by a vertex, and each arc $a_{i,j} \in A$ denotes that processor i tests processor j for $i,j \in V$. An arc $a_{i,j} \in A$ is given weights 0(1) if the testing processor $i \in V$ finds the tested processor $j \in V$ fault-free(faulty). If $a_{i,j}$=0(1) then it is said that i has a 0-link(1-link) to j. The collection of all such test weights for a particular system S is called the syndrome W formed by the test graph G of the system S. The system S is diagnosed through the examination of the syndrome W to locate a unique set of faulty

processors F.

To facilitate the discussion of this testing model, the following sets are defined. For each vertex $i \in V$, let

$$\Gamma(i) = \{j | a_{i,j} \in A\}$$

$$\Gamma^{-1}(i) = \{j | a_{j,i} \in A\}.$$

Discussion of fault diagnosis through testing must consider a *fault model* of deviant processor behavior. In the PMC model [PrMC67], the following behavior is assumed. If processor i is fault-free and tests processor j, a fault-free (faulty) processor, then $a_{i,j} = 0(1)$. If processor i is faulty, then the test outcome is unreliable. This is summarized in Figure 5.1(a). The term *symmetric invalidation* is used to describe the PMC model. The characterization of diagnosable graphs G given in [HaAm74] is a function of the maximum number of tolerable faults and the connection assignment. If a system S obeys this characterization, then it is said to be t-fault diagnosable - the system can withstand up to t faults and diagnosis can still identify all the faulty units in the system for all syndromes that can result from testing under the PMC model. Performance of the diagnosis is described by the same terms correct and complete from Chapter 3. [DaMa84] presents an algorithm which can diagnose a system S in $O(n^{2.5})$ time. The diagnosis provided by this algorithm is both correct and complete as the fault set F is exactly the set of faulty processors.

The PMC model can be unrealistic since it requires a processor to always identify itself to a non-faulty testing processor. This may not occur if the test itself is "incomplete," that is, a faulty processor can pass the test but still be faulty. The model is also unrealistic in that it requires that a faulty processor exhibit only permanently faulty behavior. The CMOS technology involved requires consideration of intermittently faulty behavior. To review the distinction between these two cases presented in Chapter 1, once a permanently faulty unit fails, it suffers a hard failure - it never recovers. An

intermittently faulty unit may function incorrectly and then correctly at a later time with the possibility for future failure/recovery cycles.

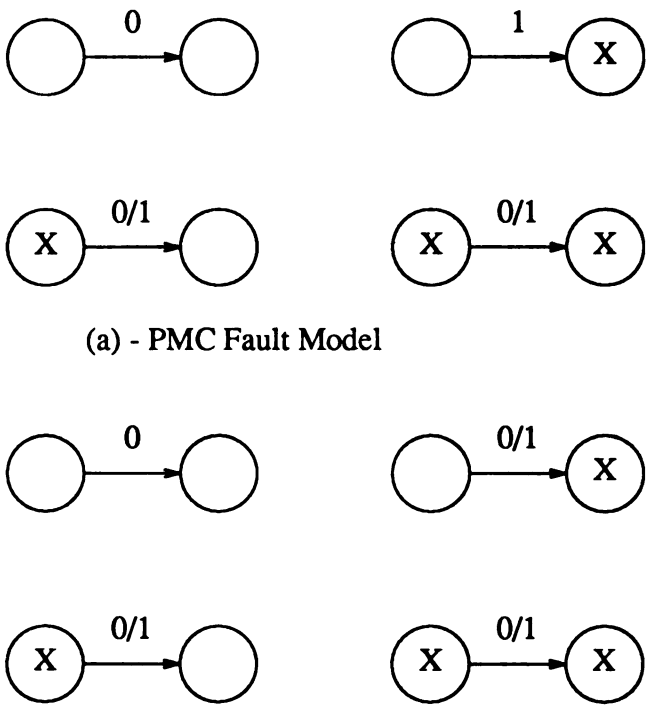

Figure 5.1. Fault Models

Fault diagnosis under the intermittent fault model has received much study [MaMa78, YaMa86b]. For intermittently faulty behavior, if i is a fault-free processor testing a faulty processor j, the outcome is unreliable. All other conditions of the PMC model remain true. This is depicted in Figure 5.1(b). This model is also characterized by symmetric invalidation.

Let $/V_1/$ be the cardinality of set V_1.

Lemma 5-1 [MaMa78]: A system S is t_i-diagnosable if and only if for any set of processor's $V_1 \subseteq V$ where $0<|V_1|<t_i$, we have that

$$|\bigcup_{i \in V_1} \Gamma^{-1}(i) - V_1| > t_i.$$

If a system S subject to intermittent faults is characterized as t_i-fault diagnosable, then a correct diagnosis resulting in identification of a unique fault set F may be obtained provided that at most t_i processor's fail. [YaMa86a] presented an algorithm for t_i-fault diagnosable systems which yields a unique fault set F. However, under certain characterizable syndromes, the algorithm may not achieve a complete diagnosis. Thus, based on the syndrome that occurs, faulty units may *escape* detection. This work provides a characterization of syndromes for which their algorithm produces an incomplete diagnosis. To describe this characterization, we adopt their notation. For each $i \in V$, the 0-ancestors of i correspond to the set

$$A_0(i) = \{j | a_{j,i} = 0\}$$

Theorem 5-1 [YaMa86a]: A faulty processor j is detectable by Yang and Masson's algorithm if and only if

$$|A_0(j) \cup \{j\}| \leq t_i$$

where t_i is the maximum tolerable number of intermittent faults.

Theorem 5-1, however, is not a full characterization of t_i-diagnosability. Consider the testing digraph G and syndrome shown in Figure 5.2 for $t_i = 1$. The following discussion shows that processor 2 is the only faulty processor. If processor 2 is not faulty, then assume processor 0 is faulty as indicated by the testing link $a_{2,0} = 1$. But then either processor 1 or processor 2 must also be faulty since a 1-link exists between them. Since processor 2 is fault-free, then processor 1 must be faulty and the 1-link from processor 1 to processor 2 ($a_{1,2}$=1) is an erroneous test result reported by processor 1. This, however, is

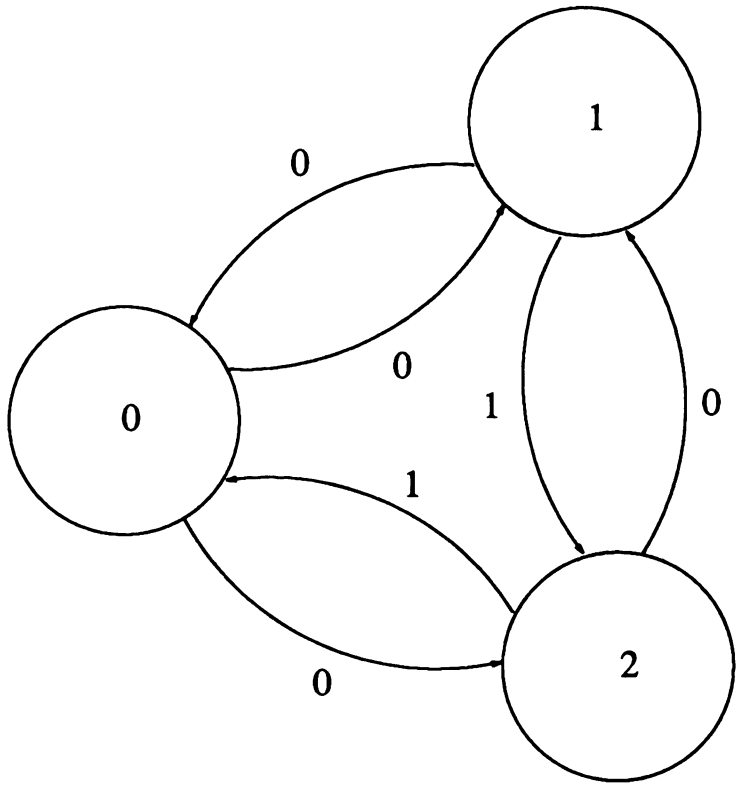

Figure 5.2. Syndrome Undiagnosable by Yang and Masson Algorithm

t_i-diagnosable system with testing digraph G.

processor 2 is faulty, $t_i = 1$

a contradiction since now the fault set F has cardinality 2 ($|F| = |\{0,1\}| = 2 > t_i$) and exceeds the bound on the number of faults. If we assume processor 1 is faulty, a similar analysis reaches the same contradiction - the syndrome contains too many faulty processors. Thus, the only possible diagnosis is that $F = \{2\}$. It is easily verified that the testing digraph G satisfies Lemma 5-1 for $t_i = 1$ and is thus t_i-diagnosable. However, $A_0(2) = \{0\}$ which when applied in Theorem 5-1, yields $|A_0(2) \cup \{2\}| = 2 > t_i$. Thus Yang and Masson's algorithm cannot detect the faulty processor in this syndrome even though it is diagnosable. The reason for this should be clear. The Yang and Masson characterization of diagnosable syndromes only considers detection of faulty behavior

through implication of the suspected faulty processor. The analysis just performed used the additional information of faulty implication by the suspected faulty processor.

This prompts the question of a whether a better diagnosis algorithm exists. Optimally, a diagnosis algorithm that can perform a complete diagnosis is desired. We now show that an incomplete diagnosis is the best that can be achieved, and that a complete diagnosis for all syndromes is impossible.

Definition 5-1: A *Non-masking Diagnosis* is one in which the result of a test $a_{i,j}$ from processor i to processor j cannot be discarded by the diagnosis algorithm unless it is known that processor i is faulty.

Definition 5-2: A *Deterministic Diagnosis* is a diagnosis in which no individual probabilities are assigned to either the failure of the processors nor to the results of the tests.

An non-masking diagnosis cannot, as in [GuRa86], find a processor i' faulty based on some minimal cardinality proper subset of $\Gamma^{-1}(i')$. To do so establishes a diagnosis based on repetitive testing. Deterministic diagnosis forces symmetric invalidation of two processors i' and j' which share an incident 1-link. The diagnosis algorithms given by [YaMa86a, DaMa84] are both deterministic non-masking diagnosis algorithms.

Lemma 5-2: Given any testing digraph $G(V,A)$ there exists one syndrome, for which any non-masking diagnosis is incomplete for even the single intermittent fault case.

Proof: Let $G(V,A)$ be a digraph with weighted arcs $a_{i,j}$ as described in Section 5.1. Let i' be a faulty vertex with $a_{i,i'} = 0$ for $i \in \Gamma^{-1}(i')$ and $a_{i',j} = 0$ for $j \in \Gamma(i')$. Then it is impossible to diagnose the system completely since i' cannot be assigned to the set of faulty units F. □

Thus a faulty processor may pass all tests and report pass for processors it tests in the trivial case. The syndrome described by Lemma 5-2 is called the *trivial syndrome*. However, even if the faulty processor manifests itself by assigning 1-links to some incident edges, it can still escape detection as the following theorem shows.

Theorem 5-2: Given any testing digraph $G(V,A)$ with $|A| \geq 1$, there exists at least one syndrome, exclusive of the trivial syndrome, for which any deterministic, non-masking diagnosis is incomplete for even the single intermittent fault case.

Proof: Let $G(V,A)$ be a testing digraph with weighted arcs $a_{i,j}$. Without loss of generality, let i' and j' be given such that $j' \in \Gamma(i')$ and $a_{i',j'} = 1$. Furthermore let $a_{i,i'} = 0$ for $i \in \Gamma^{-1}(i')$ and $a_{j',j} = 0$ for $j \in \Gamma(j')$. Note that one or both of the sets $\Gamma^{-1}(i')$ and $\Gamma(j')$ may be the empty set. Since the diagnosis is deterministic, symmetric invalidation states that there is no conclusive information to favor one processor as faulty over the other. Thus it is impossible to diagnose the system as either processor i' or processor j' may be in the set of faulty units F. □

Further lessening of the restrictions on the intermittent fault model leads to the most general class of faults, the Byzantine fault model. In this model, a faulty processor no longer exhibits random behavior when tested as in the intermittent faulty mode, but assumes a malicious personality. By this malicious behavior, a faulty processor will perform the most disruptive function at the most critical time. This is clearly an attractive fault model since any algorithm tolerant of Byzantine faults is tolerant of all faults.

In the Byzantine case, the syndrome testing fault model is exactly the same as the intermittent fault model (in terms of testing arc weights). However, the philosophy of the assignment is different. In the syndrome diagnosis, the faulty units will *elude* detection. In light of the Lemma 5-2, a Byzantine processor can always completely elude detection by passing all its tests through reporting of the trivial syndrome. Furthermore, it will confuse the diagnosis, as in Theorem 5-2, such that an incomplete diagnosis is obtained.

Two main points are stressed in this discussion. The first is that the set of t_i-diagnosable syndromes given by [YaMa86a] is a proper subset of the true set of t_i-diagnosable syndromes. More importantly, we have shown that there can never be a diagnosis algorithm for t_i-diagnosable systems which is always complete. This is particularly important when the Byzantine class of faults is considered. Unlike algorithms

which always function correctly in the presence of Byzantine faults, such as Byzantine Agreement, no algorithm can perform Byzantine diagnosis completely. Numerous algorithms have been proposed that function in the presence of Byzantine faults and purport diagnosis of Byzantine faults [GuRa86, ShRa87]. In reality these algorithms are tolerant of intermittent failures as they require repeated testing to uncover faulty behavior. A true Byzantine fault will never manifest itself in a manner diagnosable by these algorithms.

5.2 BYZANTINE GENERALS PROBLEM

The Byzantine Generals problem [LaSP82] is a rephrasal of [PeSL80] giving the problem a more colorful name and description. In the Byzantine Generals scenario, the Byzantine army is encamped outside of an enemy city. Each division has a general (resource) some of whom are traitorous. The solution requires that the loyal generals decide on the same plan of action ("attack" or "retreat") and that a small number of traitorous generals cannot force adaptation of a bad plan (a bad plan being that some generals attack while others retreat). Note that for all the loyal generals to decide upon the same plan of action, all the loyal generals must obtain the same information. This is done by the exchange of messages. A traitorous general may send different values to different generals. In order to prevent a small number of traitors forcing the adoption of a bad plan, the value sent by a loyal general must be used by every loyal general as the correct value. Furthermore, all loyal generals must use the same value from a traitorous general.

An example adapted from [PeSL80] for the four general, one traitor, case is given in Figure 5.3. For the complete algorithm, the reader is encouraged to consult one of the two above references. We now substitute processors for generals and use multiple values instead of the single valued "attack" or "retreat."

In Figure 5.3 there are four processors ($P_1 - P_4$), one of which is faulty (P_1). It is not known, however, by P_2, P_3, or P_4, that P_1 is faulty. Nor may P_1 be aware of its

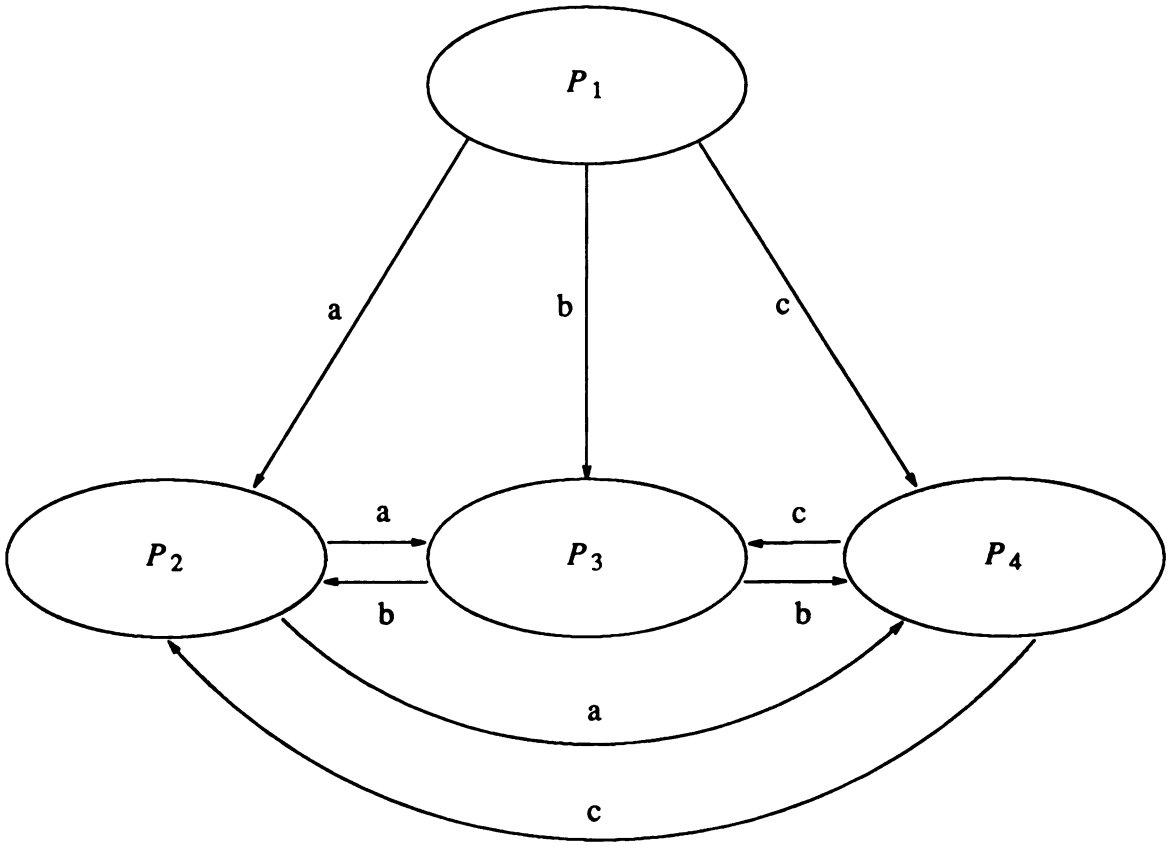

Figure 5.3. Byzantine Agreement (P_1 is faulty)

(from [LaSP82])

faulty condition. If the faulty processor P_1 broadcasts over point to point links, it is free to send different values to different recipients of its broadcast. Say it sends the value "a" to P_2, "b" to P_3, and "c" to P_4. Each processor, with its own local view of the system, has no idea that the other processors have received different values for the same broadcast message. By exchanging values, it is possible for each processor to relay the value it received from P_1 to each of the other processors. Thus if each one of the receivers relays its version of what P_1 sent to the other receivers, each receiver will have three versions of P_1's value. From the point of view of one processor, say P_2, it will receive one version (the value "a") directly from P_1, the value "b" from P_3 as P_1's value and the

value "c" from P_4 as P_1's value. Since all receivers have received different values in this case, a strategy might be to pick the lowest ASCII coded value, say "a." Thus P_2 will use the value "a" as P_1's value. Since both P_3 and P_4 perform the same computation as P_2, and since P_2, P_3, and P_4 are non-faulty, they all come to the same decision on P_1's value, namely "a." An algorithm which allows the non-faulty processors to come to the same agreement on a value is said to solve the *Byzantine Generals Problem.*

A *round* is a set of message exchanges. The above example used two rounds of information interchange. [DoSt82] shows that for n generals with t faulty, $t+1$ rounds of interchange are needed to reach agreement among $n \geq 3t+1$ generals. If either less than $3t+1$ generals or fewer than $t+1$ rounds are employed, no reliable consensus can be reached. This solution is known to be *t-resilient*, that is, it can withstand t faults. This is the *Unauthenticated Byzantine Generals* solution. It requires three assumptions.

1) Every message that is sent is delivered correctly.
2) The receiver of a message knows who sent it.
3) The absence of a message can be detected.

The *Authenticated Byzantine Generals* solution requires, in addition to assumptions 1-3 above, the following assumption.

4) Each loyal general can send unforgable signed messages.

For the authenticated solution, a three general solution exists and the protocol is known to be *n-resilient*, that is, it can withstand $n-2$ faults where n is the number of generals in the system (n must be >2 or the problem is vacuous).

5.2.1 Synchronic Algorithm Constraints

In voting, we are concerned with whether the algorithm is *synchronous, asynchronous,* or *partially synchronous* characterized by bounds Δ and Ψ. Δ is the bound on the time to send a message from one processor to another, and Ψ is the maximum clock drift between any two processors.

5.2.1.1 Synchronous Algorithms

If Δ and Ψ are known *a priori*, then the system is said to be synchronous. The Byzantine generals problem can be solved for t-resiliency or n-resiliency in the following cases:

1) Synchronous processors, synchronous communication
2) Synchronous processors, synchronous message order
3) Broadcast transmission and synchronous message order

Synchronous algorithms, such as used in the FTMP [HoSL78], do not generally fit the DMMP model. However, specific applications may indeed fit this model.

5.2.1.2 Asynchronous Algorithms

In an asynchronous system, neither Δ nor Ψ exist. It is impossible to find consensus in an asynchronous environment for even one faulty Fail-Stop processor [FiLP85]. There exists a window of vulnerability in which a faulty processor can indefinitely delay agreement. Deterministic asynchronous Byzantine Agreement can be achieved only if processors do not fail during the protocol execution. However, other, more realistic possibilities, also permit asynchronous agreement.

[Perr85], building on the work of [Rabi83], proposed a randomized Byzantine Generals algorithm which reaches agreement with probability $1-2^{(-R)}$ in R rounds using the concept of "Shared Secrets" [Sham79] in which Digital Signature Authentication is employed and a *trusted dealer* gives out public key encripted data. Shared secrets are used to circumvent the impossibility result given in [FiLP85]. The reconstruction of the secret avoids a process from indefinitely waiting for messages from faulty processors. This algorithm will achieve agreement with probability 1 if $R \rightarrow \infty$. The expected number of rounds is four; two rounds are needed to reach agreement and two rounds are needed to verify the proof.

This algorithm works on the following principle. A station wishing to enter into agreement enters into a loop bounded by R. This loop consists of procedures Poll, Lottery, and Decision. Poll sends its encrypted message to all stations. It then reads all values coming into it from this round. Collecting $n-t$ values, it decides the plurality vote and the number of stations voting this way. The station then enters the Lottery phase. The Lottery asks for the secret message from all processes for this round. The secret is then decoded when t different messages have arrived. The final phase is the Decision. Each station has its own idea of what the message should be. Based on the randomly generated secret just received, the message is either accepted or rejected. This procedure continues until acceptance is met or R is reached. Notice that since we have the completely asynchronous case, messages may arrive out of order, be delayed indefinitely, or exhibit other Byzantine manifestations. Thus each round of the lottery should get its messages from some asynchronous process which collects messages for all rounds regardless of the current round. [Rabi83] used signed messages in his algorithm, but [Perr85] showed that this was unnecessary in agreement with the theoretical result given in the Byzantine Generals problem statement. In both cases, the resulting algorithms tolerate t faulty processors where $n>6t$.

The method of shared secrets is limited. A trusted dealer must exist to reliably predistribute the shared secret values before each process begins. Additionally, it must be assumed, although not explicitly said in either paper, that a process does not know the secret if and only if it sends a bad message.

An alternative to algorithms employing shared secrets is presented by [DLPS86] in which approximate agreement can be reached in an asynchronous environment. This algorithm handles Byzantine Agreement with successive approximations converging to some $\varepsilon \geq 0$. At first it is unclear how one could use such an approximate algorithm. However, it becomes useful when one considers the problem of synchronizing clocks or of agreeing on real valued input from data collection sensors. The algorithm functions in a

number of rounds indirectly bounded by the requested ε. At each round, $n-t$ values are requested from other processors. A selection algorithm chooses the smallest $2t$ elements and then takes the average. When the average is less than ε, the algorithm terminates. This algorithm tolerates t faulty processors where $n>5t$. This procedure assumes that all processes terminate, but not necessarily at the same time. It also tolerates a greater proportion of faulty processes than $5t$ because it handles processes which exhibit transient Byzantine behavior; that is, they fail and then recover to a correct mode of operation.

5.2.1.3 Partially Synchronous Case

A partially synchronous system is one in which both Δ and Ψ hold eventually but are not known *a priori*. Agreement in the partially synchronous envrionment is solvable without authenticated signatures [DwLS88]. The algorithm tolerates t faults where $n>3t$. The algorithm locks and unlocks various supposed values of agreement for each round. If a locked value is later learned to be not an agreed upon value, it is unlocked and the algorithm continues. The algorithm is guaranteed to terminate, however, the number of rounds is specified as polynomially in N, Ψ, and Δ which for an actual implementation is expensive.

5.2.2 Masking vs. Detection

The Byzantine Generals problem has been the subject of study as a solution to the problem of finding *a consensus* among a number of processors in a faulty environment. The *weak consensus* problem is solved if when the local value of all non-faulty processors is v and no processors fail, all processors reach the same agreement, namely v. The *strong consensus* problem adds that all non-faulty processors reach agreement on v in the presence of up to a predetermined number of faults denoted by t.

While Byzantine Generals solution algorithms mask faults, they alone have a drawback in the solution of the fault-detection problem. In the introductory example it was

clear that processor P_1 was faulty, since all receivers relayed different values. If the sender is non-faulty, however, and a receiver is faulty and thus relays the wrong values, processors may erroneously flag P_1 as faulty when, in reality, some other processor is faulty. The Byzantine Generals solution cannot implicitly locate the faulty processor (Byzantine detection is covered in Section 5.2.4). It can only guarantee that a consistent state of the system can be produced for each processor. However, an acceptance test run on this view by each non-faulty processor will produce the same set of faulty processors as a test result.

5.2.3 Vector Byzantine Agreement *Agree*

Byzantine Agreement (even in the best case of synchronous behavior) is expensive. The problem with each of the agreement algorithms presented above is that to achieve the necessary concensus for application oriented fault-detection among n processors, n individual agreements are necessary. Vector Byzantine Agreement collapses these into a single agreement by trading off communication complexity for time and space complexity. Since message initiation is the most expensive part of the communication, lowering the number of agreements lowers the number of messages.

The algorithm presented achieves Vector Byzantine Agreement under the assumptions of synchronous processors and communications and a reliable completely connected (within the agreement group) communication network. These are not unreasonable assumptions. If communication network errors occur, they may be charged to one of the processors in the agreement. The completely connected communication requirement only extends to the members of the agreement group which is typically of small size and is seen to be easy to construct for the Poker environment [Snyd84]. Furthermore the algorithm extends to the less than completely connected case using the result of [Dole82]. The concept of Vector Byzantine Agreement is not limited to synchronous processors. Any of the partially synchronous [DLPS86] or asynchronous [DwLS88]

[Perr85] algorithms presented above could be adapted to the form of Vector Byzantine Agreement to function as a distributed diagnostic basis.

A constraint predicate Φ may be applied by each non-faulty processor to its local copy of the agreed upon vector. The predicates will be constructed in such a way that each non-faulty processor will obtain the same decision as every other non-faulty processor.

5.2.3.1 Agree

The algorithm *Agree* described in this section achieves Vector Byzantine Agreement.

We introduce some notation at this point to formally describe *Agree*.

n In the agreement, there are n processors. The ith processor is denoted P_i for $i=1,...,n$.

t Among the n processors, the maximum number of tolerated faults is t.

v_i Each processor that enters into the agreement begins with a local value that it wishes to broadcast to other processors in the agreement. For P_i, this value is denoted by v_i. The value v_i is treated as correct by P_i. Note that v_i may or may not be equal to v_j if $i \neq j$.

V^i At the successful termination of the agreement protocol, each processor has obtained a vector of values. Each component of this vector corresponds to the agreed upon local value of a processor in the agreement. For processor P_i, V^i denotes P_i's vector of values. Each component of this vector is the value $V^i(j)$ where $j=1,...,n$. $V^i(j)$ denotes the value that P_i uses as P_j's local value v_j. It is the case that $V^i(i) \equiv v_i$ if P_i is non-faulty.

Agree departs from typical Byzantine agreement algorithms [LaSP82, Dole82] which deal with single valued agreement and instead reaches a consensus on the entire vector of local values in the same set of rounds. Thus at termination, each non-faulty

processor will return its vector V^i which achieves *Vector Interactive Consistency Conditions.*

Vector Interactive Consistency Conditions:

VIC1: Any two non-faulty processors P_i and P_j obtain vectors V^i and V^j such that $V^i(k) = V^j(k)$, k=1,...,n.

VIC2: If a processor k is non-faulty with local value v_k, then for any non-faulty processor P_i, $V^i(k) = v_k$.

The above two conditions may seem the same, however, they are not. Condition (VIC1) states that any two non-faulty processors obtain the same copies of the local values of all the processors in the agreement. However, these values may or may not be equal to the local value v_i that a particular processor P_i actually holds. Condition (VIC2) states that all non-faulty processors agree on the same value as a particular processor P_k's local value v_k if the particular processor is non-faulty.

The other protocols mentioned at the beginning of this section require $n(t+1)$ rounds to reach agreement as each value must be agreed upon individually before the next round can start. Our protocol achieves agreement in $t+1$ rounds.

The protocol recursively broadcasts and then negotiates on the broadcast values. Each negotiation requires a message exchange, and thus each processor recursively enters into agreement on the exchanged messages.

The algorithm *Agree* shown in Figure 5.4 achieves Vector Interactive Consistency (Reference Theorem 5-3 which follows). Each processor running *Agree* will send its local value to all other processors. Then *Agree* is called recursively to reach agreement on these values. The algorithm tolerates a maximum number of faults, t. It will be shown in Theorem 5-3 that agreement can be achieved if $n > 3t$. Initially processor P_i, $1 \le i \le n$, invokes the algorithm *Agree* with *Agree*$(0,v_i)$. Let Q be the set of processors in the agreement excluding the calling processor.

Agree is called recursively. The first time each P_i's v_i is broadcast to all other processors. Subsequent calls reach agreement on the received broadcast values. Thus, $Agree(0,T_0)$ performs the initial broadcast of each v_i. $Agree(1,T_1)$ calls for agreement on the values received in $Agree(0,T_0)$ by invoking $Agree(2,T_2)$.

Let T_m be an m-dimensional square array in which each dimension is of length n. In each call to $Agree(m,T_m)$ each processor sends its own T_m and assembles $n-1$ m-dimensional square arrays from the other $n-1$ processors to form an $(m+1)$-dimensional T_{m+1}.

Each T_m from P_k is indexed in the $(m+1)$-dimensional $T_{m+1}(k)$. Similarly, each m-dimensional T_m is indexed in the $(m+2)$-dimensional sqaure array T_{m+2} by $T_{m+2}(j,k)$ where j indexes a T_{m+1} and k indexes the T_m within that T_{m+1}. In the proofs, processor P_k's T_m is referenced by T_m^k. Where no ambiguity exists, the superscript on T is dropped.

Let R represent the indices of elements in the m-dimensional square array T_m: $i_1, i_2, ..., i_m$, where $1 \le i_j \le n$, for $1 \le j \le m$. Thus an individual element of T_m is referenced by $T_m(R)$. Similarly define $T_{m+1}(k,R)$ to be an element in the k'th m-dimensional square array T_m and $T_{m+2}(j,k,R)$ to be an element in the k'th T_m of the j'th $(m+1)$-dimensional T_{m+1}.

The function *majority* returns the majority value of its arguments (the individual elements of each array). If no majority exists, it returns the lowest ASCII coded value. Note that this is a purely arbitrary choice; however, it will be made consistently by all non-faulty processors. If no value is received for a particular element, *majority* chooses an arbitrary (but consistent) value to use.

Notice that at the initial call, the T_0 is a scalar. At termination of $Agree(0,T_0)$, a vector T_1 is returned satisfying conditions VIC1 and VIC2.

Algorithm *Agree*(m,T_m);

allocate storage $T1_{m+1}$;

1) $T1_{m+1}(id) \leftarrow T_m$; /* id is the calling processor id */
2) **for all** $P_k \varepsilon Q$, **send** T_m **to** P_k;
3) **for all** $P_k \varepsilon Q$, **receive** an m-dimensional square array **from** P_k and **store in** $T1_{m+1}(k)$;

if $t>m$ **then**

allocate storage $T2_{m+2}$;

4) $T2_{m+2} \leftarrow$ *Agree*$(m+1,T1_{m+1})$
5) **for each** $k \varepsilon Q$
for each R such that $i_g \neq i_h$ for $1 \leq g,h \leq m$ and $g \neq$ h
for each $j \varepsilon Q, j \neq i_g$ and $j \neq k$ for $1 \leq g \leq m$
$T1_{m+1}(k,R) \leftarrow$ **majority** $(T1_{m+1}(k,R),T2_{m+2}(j,k,R))$;

return$(T1_{m+1})$;

Figure 5.4. Algorithm Agree

5.2.3.2 Example of Algorithm *Agree*

A short example demonstrating the use of *Agree* in presented below. Figure 5.5 contains an example of the algorithm with $n=4$ and $t=1$. Processors are numbered P_1–P_4 with P_3 as the faulty processor.

Initially, each P_i invokes algorithm *Agree*$(0,v_i)$ $(v_i \equiv T_0^i)$. At Step 1, each processor sends T_0 to the other three $(3=n-1)$ processors. Each processor then receives three values in Step 2. These values, along with the local processor's v_i, compose the square array $T1_1$, a one dimensional square array. This is shown as a 4 element vector in Figure 5.5. Row indices index the received local value from each processor. Since P_3 is faulty and exhibits Byzantine behavior, it may send any values it wants or may send no value at all.

Agree$(1,T1_1)$ is recursively called in order to exchange messages on each receiver's view of the initial broadcast. *Agree*, in Step 2, sends the square array (T_1) to the three other processors. In Step 3 it receives the three other 1-dimensional square arrays. These

three 1-dimensional square arrays along with the square array T_1 it received from *Agree* in the previous recursion are used to create the 2-dimensional square array $T1_2$. This is shown in Figure 5.5 as a two dimensional array. The row indices index the received $T1_1$ for each P_i. The "?"'s for P_3's $T1$ indicate "don't care" values since P_3 is faulty and nothing can be assumed about its internal values. Since now $t=m=1$, $Agree(1,T_1)$ returns $T1_1$ to $Agree(0,T_0)$.

In $Agree(0,T_0)$, each processor applies its majority function to the returned 2-dimensional square array $T2_2$ and to its $T1_1$ for each of the other processors. Processor P_1 obtains the majority values in the following way:

$$T1_1(2) \leftarrow majority(T1_1(2), T2_2(3,2) T2_2(4,2)) = majority(b,b,b)$$

$$T1_1(3) \leftarrow majority(T1_1(3), T2_2(2,3) T2_2(4,3)) = majority(a,b,c)$$

$$T1_1(4) \leftarrow majority(T1_1(4), T2_2(2,4) T2_2(3,4)) = majority(d,d,c)$$

Thus P_i returns [*abad*] as its vector. Note that in all cases, $T1_1 \equiv T2_2(k)$ in every non-faulty processor P_k's computation. Thus the majority could have been taken over $T2_2$.

The majorities above decide the final square array $T1_1 = V^i$ for the non-faulty P_i, $i=1,2,4$. Each processor runs majority 3 times, once to decide on each of the other processor's value. Notice that P_i does not decide upon its own value T_0^i. This is known by P_i to be correct.

Processor P_3 has attempted to thwart a consensus by sending different values to different processors. However each non-faulty processor runs *majority* on the same set of values. Here we have chosen to choose the "lowest" value (ASCII collating sequence) for the majority if no majority value can be found. Thus each non-faulty processor uses "a" as the value for P_3.

Vectored interactive consistency is seen to be achieved as each non-faulty processor i uses the vector of values $V^i = [a,b,a,d]$. Each processor uses the communications medium to send six messages in two rounds. The total space requirement for this case is $O(n^2)$. In general, the algorithm *Agree* requires $O(n^{t+1})$ space and computation

complexity and a communication complexity of $(n-1)(t+1)$ messages in $t+1$ rounds of information exchange.

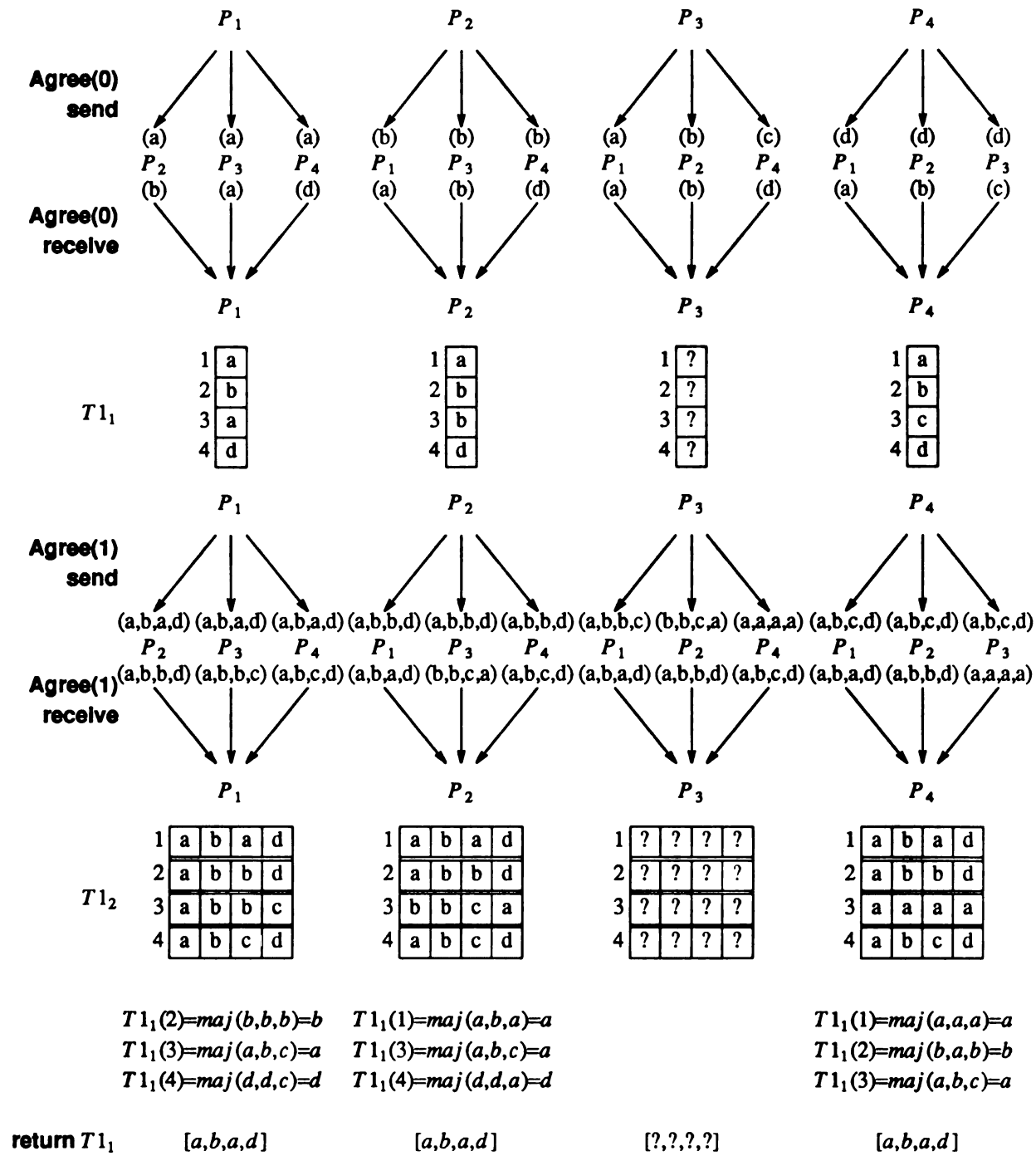

Figure 5.5. Example of Algorithm *Agree* for $n=4, t=1$.

5.2.3.3 Proof of Correctness

It was stated that *Agree* achieves VIC1 and VIC2 for $n > 3t$. We must first prove that our algorithm satisfies vectored interactive consistency condition (VIC2) for a weaker condition of fault tolerance. This will be necessary when we consider the case of deciding on a vector when the transmitter of that vector is reliable and its recipients contain all the faults. This condition violates our intentions of achieving VIC1 for $n \leq 3(t-m)$ for a given recursion m.

Lemma 5-3: *Agree* satisfies condition VIC2 for $n > 3t-m$ where $t \geq 0$ is the maximum number of faulty processors and m is the index of recursion.

Proof: We proceed by induction on $t-m$. For each correct sender in Step 2, all correct receivers receive each correct sender's value. If $(t-m)=0$, *Agree* fulfills VIC2 because the number of faults is zero and every receiver uses the sender's value.

Assume that *Agree* works for $(t-m)-1$, we wish to show it is correct for $(t-m)$, $t>m$.

1) In $Agree(m,T_m)$, each correct receiver receives $n-1$ square arrays T_m into $T1_{m+1}$ at Step 3 from the $n-1$ calls to **send** at Step 2 done by the $n-1$ other participating processors.

2) Each correct receiver then applies $Agree(m+1,T1_{m+1})$. The number of processors considered in Step 5 (the number of individual elements of the square array $T1_{m+1}$) in the agreement is reduced by 1. Thus *Agree*(m+1) runs with values from $n-1$ processors (1 directly obtained from the sender and $n-2$ relayed by other receivers for each component of the vector $T1_{m+1}$).

3) Since $Agree(m,T_m)$ is called with n square arrays T_{m-1} , and by hypothesis, $n > 3t-m$, then $n-1 > 3t-(m+1)$. Thus by the inductive hypothesis and by (2),

$Agree(m+1,T1_{m+1})$

achieves VIC2 with $n-1 > 3t-(m+1)$. Thus each non-faulty processor receives the same $T2_{m+2}(k)$ for each non-faulty P_k.

4) To achieve a majority vote on $n-1$ components of $T2_{m+2}(j,k,R)$ such that the majority is governed by values reported from non-faulty processors, it must be true that: $n-1-t > t$ or $n-1 > 2t$. Since there are at most t faulty processors and $t > m$,

$$n-1 > 3t-m-1$$

$$n-1 > 3t-(t-1)-1 \geq 2t.$$

Each non-faulty processor has enough reported values from non-faulty processors in $T1_{m+1}$ equal to $T_m^k(R)$ to compute the same majority value for any non-faulty P_k, namely T_m^k. Thus for any non-faulty processor, $T1_{m+1}(k)=T_m^k$ for each non-faulty P_k and VIC2 is satisfied. □

Theorem 5-3: $Agree(m,T_m)$ achieves vectored interactive consistency VIC1 and VIC2 for $n>3(t-m)$.

Proof: We proceed by induction on $(t-m)$.

If $(t-m)=0$, since there are no faulty processors under consideration, each P_i receives, in Step 3, the sender's local value T_m^j for any P_j (sent in Step 2). Each P_i then returns the same vector $T1_{m+1}$.

Assume that $Agree(m+1,T_{m+1})$ achieves VIC1 and VIC2. We want to show that this is true for $Agree(m,T_m)$, $t > m$

Case 1 The sender in Step 2 is non-faulty. By Lemma 5-3, each non-faulty $P_i \varepsilon Q$ receives the $T_m^k=T1_{m+1}(k)$ for each non-faulty P_k. By taking $m=0$ in Lemma 5-3, we have $n > 3t$. Since the sender is non-faulty, then VIC1 follows from VIC2.

Case 2 The sender in Step 1 is faulty. Thus each processor receives possibly a different value, the wrong value, or no value at all. Each correct receiver

executes $Agree(m+1,T1_{m+1})$. Since one of the faulty processors is known to be the sender, only $(t-m)-1$ faulty processors are the receivers.

Thus since:

$$n > 3(t-m)$$

$$n-1 > 3(t-(m+1))$$

we apply the induction hypothesis to determine that each $Agree(m+1,T1_{m+1})$ achieves vectored interactive consistency VIC1 and VIC2. Thus for any two correct processors for each k and for all j, $T2_{m+2}(j,k)$ are identical. Thus each correct processor obtains the same $T1_{m+1}(k)$ in Step 5. □

We now give the time and space complexity for algorithm *Agree*.

Theorem 5-4: The algorithm *Agree* with n processors and at most t faults requires space $O(n^{t+1})$.

Proof: The algorithm is recursively called with m, $0 \le m \le t$, as the index of recursion.

For $m=0$, *Agree* requires $1 + n^1 + n^2$.

For $1 \le m < t$, *Agree* requires an additional $n^{m+1} + n^{m+2}$.

For $m=t$, *Agree* requires an additional n^{t+1}.

Summing we get:

$$1 + n^{t+1} + n(n+1)\sum_{m=0}^{t-1} n^m$$

$$=1 + n^{t+1} + n(n+1)\frac{n(t-1)}{n-1}$$

$$= O(n^{t+1}) \square$$

Theorem 5-5: The algorithm *Agree* with n processors and at most t faults has $O(n^{t+1})$ computational complexity.

Proof: The algorithm is recursively called with m, $0 \le m \le t$, as the index of recursion. For the m-th recursion, majority is performed on $(n-(m+1))^{m+2}$ values. Thus the total complexity is: $\sum_{m=0}^{t-1} (n-(m+1))^{m+2} \le \sum_{m=0}^{t-1} n^{m+2} \le \left[\frac{1-n^{t+2}}{1-n}\right] = O(n^{t+1})$. □

Theorem 5-6: The algorithm *Agree* with n processors and at most t faults has a message complexity of $(n-1)(t+1)$ messages in $t+1$ rounds of information exchange.

Proof: Each *Agree* calls itself once recursively. In each call, it sends 1 message (1 round) to each of $n-1$ processors. Since the recursion terminates when $m = t$, each processor makes a total of t calls to *Agree*, thus including the initial call, $t+1$ sends (rounds) are made for a total number of messages $(n-1)(t+1)$. □

5.2.4 Decision Metric (Predicate)

We introduce D as a formalism to represent a class of computable predicates which can be applied to the result of Algorithm *Agree*.

Theorem 5-7: Predicate $D_i \in D$ computes a decision $d \in$ {yes,no} if for any two V^kandV^l from algorithm *Agree* in any two correct processors P_k and P_l, $D_i(V^k)$ and $D_i(V^l)$ return the same decision, d.

Proof: By Theorem 5-3, all non-faulty processors P_k and P_l hold the same V such that $V^k = V^l$. Since D_i is computable, D_i returns the same decision d in each non-faulty processor. □

Relating D to the class of constraint predicates Φ of Chapter 4 completes the fault-detection model of Chapter 3. Each testing processor runs its own Φ ($=D_i$) on test input. By Theorem 5-7, each non-faulty processor reaches the same decision d and thus all non-faulty processors reach the same decision d. In this way, a reliable distributed diagnosis is achieved.

5.3 DIAGNOSTIC COMMENTS

At first glance, syndrome testing is more attractive as a diagnostic basis than Byzantine Agreement. The connectivity requirements for the intermittent fault case are lower than for Byzantine Agreement. Furthermore, syndrome testing is a diagnostic algorithm whereas Byzantine Agreement is a fault-masking algorithm. However, the incompleteness of syndrome testing and its impact in the Byzantine environment reduce the attractiveness of such a method. Additionally, all the algorithms presented in Section 5.1 require a centralized diagnostician to analyze the syndrome. Distributed diagnosis may be achieved in an elegant manner for the PMC fault model [KuRe81]. However, for the Byzantine model, reliable dissemination of the syndrome is easily seen to be exactly the problem of reaching Byzantine Agreement. Indeed, the algorithm presented by [YaMa86b], although the authors claim differently, is tolerant of intermittent faults - not Byzantine faults.

The application oriented fault-tolerance treated here makes excellent use of Byzantine Agreement as a reliable distributed diagnostic basis. Since processors involved in a calculation call upon the diagnostic basis to disseminate/receive data, each processor is guaranteed to receive the same data (ref. VIC1 and VIC2 & Theorem 5-3). Since the constraint predicate D is uniformly applied (by Theorem 5-7), the common decision is either faulty or non-faulty. If masking of a fault should occur such that the masked result appears non-faulty, it is non-faulty and no error has occurred. This can happen in the case of transient(intermittent) failures or during a failure of the communication links.

5.4 CHAPTER SUMMARY

This chapter has presented two different possible approaches to a distributed diagnostic basis. The tight coupling with the application-oriented fault-tolerance paradigm favors the use of a masking basis such as Byzantine agreement and rejects the use of syndrome testing. The algorithm presented in this chapter, Vector Byzantine Agreement,

functions as an efficient diagnostic basis for the DMMP environment under Byzantine fault conditions. Syndrome testing can never be complete in this manner, only correct.

Chapter 6

Performance Evaluation

Performance of fault-detection schemes consists of two primary components. First, expected coverage of errors is an important metric in judging the effectiveness of a testing scheme. Run time cost is the second metric in judging effectiveness since overhead dictates the usability of the scheme.

6.1 EXPECTED ERROR COVERAGE

We restate the dichotomization of error classes introduced in Chapter 3. The distributed diagnostic basis provides us with a reliable set of values on which the constraint predicate operates. The constraint predicate coverage, as we have seen from previous examples, is often not complete. Any attempt at consideration of the Byzantine model for these predicates is unreasonable. A Byzantine failure will always escape detection. Thus, instead we consider the expected coverage as our metric.

To model the error coverage we have two options. One is to generate a large number of test cases, introduce random errors, and measure the coverage. This is unattractive since it does not show which tests contribute most to the error coverage and which tests are redundant, nor does it provide much confidence in the coverage unless a large number of tests is run. The alternative is to use an analytical model. A probability model is chosen.

In general, to utilize a probability model, both a prior and posterior probability distribution of errors must be chosen. Here, however, we are only interested in the coverage

of the constraint predicate given that an error has occurred. This restricts the number of distributions to be considered to the distribution of errors given that an error has occurred. The fault model for application data errors can be chosen from some prior distribution appropriate to either the hardware faults expected, or as a function of the application itself.

Model Specifications: Define the following events corresponding to the three classes of test bases:

${}^{j}F_i^{(k)}$	Event that a result $U_j^{(k)}$ satisfies feasibility test i.
${}^{j}P_i^{(k)}$	Event that a result $U_j^{(k)}$ satisfies progress test i.
${}^{j}C_i^{(k)}$	Event that a result $U_j^{(k)}$ satisfies consistency test i.
E	Event that an error has occurred.

The absence of a particular subscript/superscript indicates that the event holds for an aggregation of the individual events. Thus the following events denoting the intersection of each class of error coverage are true when all of the tests of a particular class are true:

$${}^{j}F^{(k)} = \bigcap_i {}^{j}F_i^{(k)}$$

$${}^{j}P^{(k)} = \bigcap_i {}^{j}P_i^{(k)}$$

$${}^{j}C^{(k)} = \bigcap_i {}^{j}C_i^{(k)}$$

The absence of the superscript (k) indicates that the event is stationary over all steps k as in ${}^{j}F_i$. The meaning will be made clear in each individual context.

We are interested in the probability that the tests find an error if it has occurred. This probability is given by the following:

$$\begin{aligned} Pr(\overline{F \cap P \cap C} \mid E) &= Pr(\bar{F} \cup \bar{P} \cup \bar{C} \mid E) \\ &= Pr(\bar{F} \mid E) + Pr(\bar{P} \mid E) + Pr(\bar{C} \mid E) - Pr(\bar{F}\bar{P} \mid E) \\ &\quad - Pr(\bar{F}\bar{C} \mid E) - Pr(\bar{P}\bar{C} \mid E) + Pr(\bar{F}\bar{P}\bar{C} \mid E) \end{aligned}$$

To model the type of errors that can occur, we can choose either a discrete density function or continuous distribution function. In the former case, the discrete nature results from the discreteness of the computer word used to contain the result. The probability function of the random variable denoting the actual value of the result is the hypergeometric distribution. Assume that the range of values that a computer word can hold is $\{x | x = -L, -L+1, ..., L-1, L\}$. Then the conditional probability of the correct result (and any particular result) is $1/2L$. For any reasonable size computer word length, this value is nearly zero which is exactly what we want. In the latter case, consider the choice of the distribution of errors from some well defined distribution such as the normal. The mean of the distribution is chosen to be equal to the actual correct value of the calculation. The choice of variance can control whether erroneous values are in close proximity to the correct value or are more widely scattered. Another possibility is to choose the conditional density function to be uniform over $[-L,L]$ as in the discrete case above. In both cases, since the distribution is continuous, the conditional probability of the correct result again is zero. We are not limited to such standard distributions. If *a priori* knowledge is available concerning the probability of faulty behavior, the model can be adjusted to accommodate.

To show how the modeling works, consider the example area predicate Φ_{F^A} of Chapter 4 and its corresponding feasibility event ${}^{j}F_{i}^{(k)}$. There is only one component of the solution, so the leading superscript is dropped. Similarly, since the final value is the only one considered, the step number is dropped. If we model the errors as coming from a discrete random distribution with the random variable E_b, then the conditional probability of detecting an error is given by

$$Pr(\bar{F}|E) = Pr(E_b<0) = \frac{L}{2L} = 0.50.$$

This shows, as we already suspected, that the predicate has a low error coverage - only 50%.

In modeling the errors as a normally distributed random variable E_c, we assume that the mean is chosen to be the actual correct value of the area δ. For simplicity, let the variance σ=1. The probability of error detection is given by:

$$Pr(\bar{F}|E) = Pr(E_c<0) = Pr(E_c-\delta<-\delta) = Pr(Z<-\delta)$$

where Z has the standard normal distribution. If we choose δ to be, say, 2, then

$$Pr(Z<-2) = .0667$$

or just 7% error coverage. Clearly under this model, the predicate Φ_{F^A} is virtually useless.

In constraint predicate analysis, it is usually true that one predicate significantly narrows the range of acceptable values, and that additional predicates only make small contributions. This type of modeling can help eliminate predicates with little usefulness to increase run time efficiency at the cost of lower error coverage.

6.2 RUN TIME OVERHEAD ESTIMATION

In Chapter 2, it was demonstrated that an algorithm with no reliability could have an exponential expected run time. In this section, expectations for the run time of a reliable algorithm are given for comparison purposes. While this analysis only covers the fault-detection aspects of this problem, it is understood that a reconfiguration will take place after the fault is detected [Garc82]. This will enable the problem solution to continue in the presence of the detected errors.

General analysis techniques are presented here with specific analysis for each problem treated given in Chapters 7 through 9. The appropriate assumptions for the DMMP environment are detailed in Table 6.1. As a simplifying assumption, the problem has been partitioned such that one piece of data has been allocated to one processor. When relaxation techniques are covered in Chapter 7, performance is treated for other data partitionings.

Implementation Assumptions		
Item	Variable	Description
Scientific Word	b	60-80 bits per computation exchange.
Data Dominated Message		Message header will not dominate the message length, i.e. a single datum will dominate the message header/trailer (worst case assumption)
Message Set-up Non-trivial	S_L	Communication setup latency (time to either send or receive a message taken by a processor)
Communication Time	$C_{cNR(cR)}$	Communication time/iteration for the unreliable (reliable) algorithm
Computation Time	$C_{pNR(pR)}$	Processing time/iteration for the unreliable (reliable) algorithm
Bus width	V_B	Communication velocity of the interprocessor interconnection

Table 6.1. DMMP Assumptions

6.2.1 Communication Complexity Analysis

If we assume communication time is the dominant factor, then the analysis considers only the communication setup latency S_L and the message transmission time. Setup latency is typically expensive. For example, the iPSC hypercube has $S_L = O(1\text{ms})$ [GrRe86]. This is large compared with a typical instruction time for most computers. If we employ algorithm *Agree*, the increase in communication will be due solely to the increased message exchange and length of the agreement.

$$C_{cNR}=V_B b+2S_L \tag{6.1}$$

$$C_{cR}=\sum_{i=0}^{t}(V_B bn^i + 2S_L) \tag{6.2}$$

$$C_{cR}\approx 2(t+1)S_L+V_B bn^{t+1} \tag{6.3}$$

Now, if we assume the setup latency S_L is of order larger than the message transmission time $V_b bn^{t+1}$ equations (6.1, 6.3) reduce to:

$$O(\frac{C_{cR}}{C_{cNR}}) = t+1 \tag{6.4}$$

Thus the reliable algorithm's communication complexity is only a linear factor in $t+1$ of the unreliable algorithm.

6.2.2 Computation Complexity Analysis

Now assume that processing time is the bottleneck. The complexity components consist of the algorithm iterations, communication setup latency, and reliability calculations. The algorithm *Agree*, as noted earlier, has $O(n^{t+1})$ computational complexity. For the reliable algorithm the run time is given by

$$C_{pR} = O(n^{t+1}+2tS_L)+O(C_{pNR}). \tag{6.5}$$

The unreliable algorithm is decomposed into internal computation and message set up times.

$$C_{pNR}= O(C'_{pNR})+2S_L$$

The ratio of the computation times of the reliable algorithm to the unreliable algorithm:

$$\frac{C_{pR}}{C_{pNR}} = O\left[\frac{C'_{pNR}+2(t+1)S_L+n^{t+1}}{C'_{pNR}+2S_L}\right] \tag{6.7}$$

In the worst case, C'_{pNR} is negligible. For reasonably small values of n and t, this ratio reduces to:

$$\frac{C_{pR}}{C_{pNR}} \approx t+1 \tag{6.8}$$

Thus for a small average n, tolerant of one fault $t=1$, both the computational and communication complexity hold the following relationship:

$$C_R \approx 2C_{NR}. \tag{6.9}$$

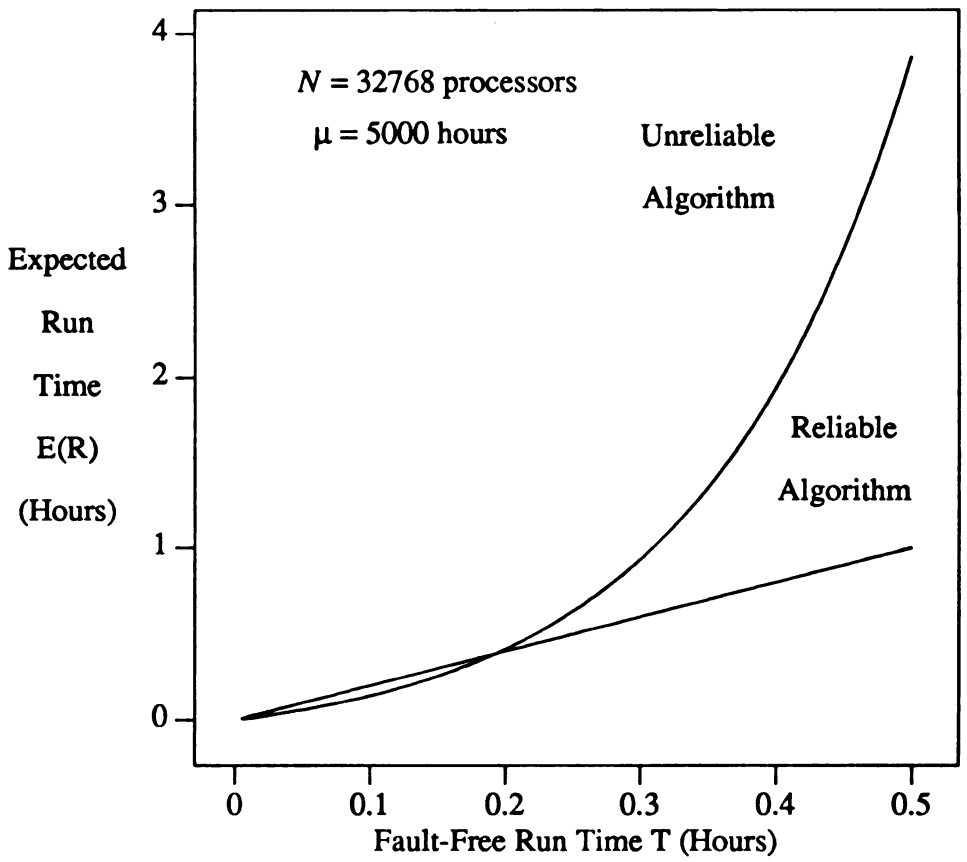

Figure 6.1. Expected run time comparison

The graph in Figure 6.1 shows the linear reliable solution compared with the exponential unreliable solution for a representative solution using the modeling techniques of Chapter 2. The point of intersection is the crossover point. To the left of this point, it is better to run with no reliability (assuming that an error is somehow detected). To the right of this point, in terms of run time, it is better to use the reliable solution. Table 6.2 shows other run time crossover points.

6.3 CHAPTER SUMMARY

Both the expected error coverage and the expected run time penalty are important metrics in judging the effectiveness of a fault-tolerant algorithm. We have seen that for problems with inadequate natural constraints, error coverage can be poor. However, for problems with good natural constraints, such as the sorting example of Chapter 4, error coverage can be 100%.

There will be a run time penalty to be paid for the use of reliability. There is a trade-off based on expected run time between using reliability or running an unreliable algorithm. For very short problems, it is better to run with no reliability than to incur the overhead of a reliable parallel algorithm. For larger problems there is a crossover point at which it is more effective to run with the reliability penalty and be guaranteed a correct solution within a bounded run time.

The next few chapters put the developed techniques to work in the reliable solution of three diverse parallel programming problems.

Failure Rate μ = 5000			
n	Crossover	*E*(*R*)	MTTF
512	12.27	24.5	4.9
1024	6.14	12.3	2.4
2048	3.07	6.1	1.2
4096	1.54	3.1	0.6
8192	0.77	1.5	0.3
16384	0.38	0.8	0.2
32768	0.20	0.4	0.1
65536	0.10	0.2	0.0

Failure Rate μ = 50000			
n	Crossover	*E*(*R*)	MTTF
512	122.70	245.4	48.8
1024	61.35	122.7	24.4
2048	30.67	61.4	12.2
4096	15.34	30.7	6.1
8192	7.67	15.3	3.1
16384	3.84	7.7	1.5
32768	1.92	3.8	0.8
65536	0.96	1.9	0.4

Failure Rate μ = 100000			
n	Crossover	*E*(*R*)	MTTF
512	245.40	490.8	97.7
1024	122.70	245.4	48.8
2048	61.35	122.7	24.4
4096	30.67	61.4	12.2
8192	15.34	30.7	6.1
16384	7.67	15.3	3.1
32768	3.84	7.7	1.5
65536	1.92	3.8	0.8

Table 6.2. Run Time Comparisons

Chapter 7

Matrix Problem and Iterative Techniques

Matrix iterative techniques have been employed since the 1940's to solve large systems of equations numerically [Varg62, Ames77]. These systems of equations arise out of the numerical solution of physical problems utilizing finite difference and/or finite element methods. Techniques of particular interest are the *pointwise relaxation* techniques, which, while not as elegant as some other methods, do contain the massive inherent parallelism necessary to take full advantage of a DMMP.

7.1 MATRIX PROBLEM SPECIFICATION

The problem is to solve the linear system $\mathbf{Au}=\mathbf{v}$, where, $\mathbf{A} = (a_{i,j})$ is a nonsingular $Q \times Q$ complex matrix, $\mathbf{v} = (v_i)$ is a complex vector, and $\mathbf{u} = (u_i)$ is the solution vector for $i,j \in \{1,2,...,Q\}$ and Q a perfect square. The method of Successive Over Relaxation (SSOR) is an iterative technique with which to obtain an approximate solution, $\mathbf{u}^{(K)} = (u_i^{(K)})$, where K is the final iteration, to this system. To parallelize an inherently sequential solution technique, a *consistent multicolor ordering* of the Q values of $\mathbf{u}^{(K)}$ (also called points) must be made. Assume $Q=N$ for simplicity and let processor $P_i, i=1,...,N$ compute value u_i. The case for $Q>N$ is treated in Section 7.4. Selection of points for computation is controlled by the iteration variable k as in the following predicate.

Definition 7-1:

$$\Lambda_{i,k} = \begin{cases} 1 & \text{if } P_i \text{ is active during iteration } k \\ 0 & \text{otherwise.} \end{cases}$$

The most common of these consistent multicolor orderings is the red/black or checkerboard multicolor ordering [Smit65] (implying, among other things, that the number of nonzero $a_{i,j}$'s per row i is no more than five). For this specific case $\Lambda_{i,k}$ becomes:

Definition 7-2:

$$\Lambda_{i,k} = \begin{cases} 1 \text{ if } (i \textbf{ div } \sqrt{N} + i \textbf{ mod } \sqrt{N}) \text{ is even and } k \text{ is even for } i \in \{1,...,N\}, k \in \{0,1,2,...\} \\ 1 \text{ if } (i \textbf{ div } \sqrt{N} + i \textbf{ mod } \sqrt{N}) \text{ is odd and } k \text{ is odd for } i \in \{1,...,N\}, k \in \{0,1,2,...\} \\ 0 \text{ otherwise} \end{cases}$$

where **div** is the integer division operator and **mod** is the remainder operator for integer division. The u_i's are assigned to points of a $\sqrt{Q} \times \sqrt{Q}$ coordinate indexed grid such that u_j's with nonzero $a_{i,j}$'s are within distance one of u_i. Each u_i is updated on odd/even values of k according to $\Lambda_{i,k}$. An odd/even iteration cycle is sometimes called two half-iterations. The remainder of this chapter considers (without loss of generality) consistent multicolor orderings of length two. The pointwise SSOR method is given by the following equation for fixed ω.

$$u_i^{(k)} = (1-\omega)u_i^{(k-2)} - \omega \frac{1}{a_{i,i}} \left[\sum_{\substack{j \neq i \\ j=1}} a_{i,j} u_j^{(k-1)} - v_i \right] \quad \text{if } \Lambda_{i,k}=1 \tag{7.1}$$

The following theorem characterizes matrices which can be solved by (7.1).

Theorem 7-1: [Varg62] Let **A** be a strictly or irreducibly diagonally dominant Hermitian $Q \times Q$ complex matrix. Then the pointwise SSOR method with $0<\omega<2$ is convergent for any initial assignment vector $\mathbf{u}^{(0)}$.

In the case of a two-dimensional grid, we can think of the values of $\mathbf{u}^{(K)}$ on the right hand side of (7.1) as neighbors of the point u_i at the compass directions North, South, East, and West. Each new value of a point is iteratively a function of the current values

of these four neighbors, the locations of which are North: (i **div** $\sqrt{Q}+1, i$ **mod** $\sqrt{Q}$), South: (i **div** $\sqrt{Q}-1, i$ **mod** $\sqrt{Q}$), East: (i **div** $\sqrt{Q}, i$ **mod** $\sqrt{Q}+1$), and West: (i **div** $\sqrt{Q}, i$ **mod** $\sqrt{Q}-1$). The program fragment of Figure 7.1 shows this typical relaxation calculation. Each processor runs the same algorithm.

Procedure Relax($u_i^{(0)}$)
/* For Processor at location P_i */
/* u_i is the data held by P_i */

$k \leftarrow 0$ /* Iteration Count */
/* u_i^0 is the initial value */

while *not* converged
  foreach (*direction* **in** {North, South, East, West})
    if ($\Lambda_{i,k}=1$)
      receive $u_{direction}^{(k)}$ from $P_{direction}$;
    else
      send $u_i^{(k)}$ to $P_{direction}$;
  if ($\Lambda_{i,k}=1$)

$$u_i^{(k+1)} \leftarrow (1-\omega)u_i^{(k-1)} - \omega \frac{1}{a_{i,i}} \left[\sum_{\substack{j \neq i \\ j \in \{\text{North,South,East,West}\}}} a_{i,j}u_j^{(k)} - v_i \right] ;$$

  $k \leftarrow k+1$;

Figure 7.1. Relaxation Skeleton

7.2 NATURAL PROBLEM/SOLUTION CONSTRAINTS

Constraint predicate development proceeds as outlined in Chapter 4. First the mathematical properties of the problem and its solution must be espoused. Theorem 7-1 allows the choice of the initial assignment vector $\mathbf{u}^{(0)}$ to be arbitrary. The next theorem shows that certain choices can facilitate constraint predicate development.

Theorem 7-2: Let **A** be a matrix satisfying Theorem 7-1 with positive diagonal and non-positive off-diagonal entries. Then the solution given by (7.1) will converge monotonically to the final solution $\mathbf{u}^{(K)}$ for the following choices of $\mathbf{u}^{(0)} = (u_i^{(0)})$ for $\omega = 1$.

1) Let $u_{\min}$ be the smallest value such that $v_i - u_{\min} \sum_{j=1}^{Q} a_{i,j} \le 0$ for all $i \in \{1,2,...,Q\}$.

   If $u_i^{(0)} = u_{\min}$ for all $i \in \{1,2,...,Q\}$, then $u_i^{(k)} \ge u_i^{(k+2)}$ for all $k \in \{0,1,...\}$.

2) Let $u_{\max}$ be the largest value such that $v_i - u_{\max} \sum_{j=1}^{Q} a_{i,j} \ge 0$ for all $i \in \{1,2,...,Q\}$.

   If $u_i^{(0)} = u_{\max}$ for all $i \in \{1,2,...,Q\}$,then $u_i^{(k+2)} \ge u_i^{(k)}$ for all $k \in \{0,1,...\}$.

Proof: For case 1), by induction on k, the iteration count.

Basis, k=0,

$$u_i^{(1)} = \frac{1}{a_{i,i}} \left[v_i - \sum_{j \ne i}^{Q} a_{i,j} u_i^{(0)} \right] \tag{7.2}$$

and since

$$v_i - \sum_{j=1}^{Q} a_{i,j} u_i^{(0)} \le 0$$

then

$$v_i - \sum_{j \ne i}^{Q} a_{i,j} u_i^{(0)} - a_{i,i} u_i^{(0)} \le 0 \tag{7.3}$$

Since $a_{i,i} > 0$, dividing (7.3) by $a_{i,i}$, simplifying and substituting in (7.2) yields

$$u_i^{(1)} = \frac{1}{a_{i,i}} \left[v_i - \sum_{j \ne i}^{Q} a_{i,j} u_i^{(0)} \right] \le u_i^{(0)} \tag{7.4}$$

Continuing to the completion of the half iteration pair,

$$u_i^{(2)} = \frac{1}{a_{i,i}} \left[v_i - \sum_{j \ne i}^{Q} a_{i,j} u_j^{(1)} \right]$$

By (7.4) and since $a_{i,j} \leq 0$, $i \neq j$; $i,j \in \{1,2,...,Q\}$

$$\leq \frac{1}{a_{i,i}} \left[v_i - \sum_{j \neq i}^{Q} a_{i,j} u_j^{(0)} \right]$$

By (7.3)

$$\leq u_i^{(0)}$$

Now assume $u_i^{(l)} \leq u_i^{(l-2)}$ for $l=2,3,...,k-1,k$. Then for $l=k+1$

$$u_i^{(k+1)} = \frac{1}{a_{i,i}} \left[v_i - \sum_{j \neq i} a_{i,j} u_j^{(k)} \right]$$

Since $a_{i,j} \leq 0$, $i \neq j$,

$$\leq \frac{1}{a_{i,i}} \left[v_i - \sum_{j \neq i} a_{i,j} u_j^{(k-2)} \right]$$

$$\leq u_i^{(k-1)}$$

The proof for case (2) is symmetric. □

Theorem 7-3: In the solution of (7.1), each $u_i^{(k)}$, $i \in \{1,2,...,Q\}$, $k \in \{0,1,...\}$ is bounded by $u_{\min} \geq u_i^{(k)} \geq u_{\max}$ for $u_i^{(k)}$, $u_i^{(k+2)}$ satisfying the conditions of Theorem 7-2.

Proof: Assume that for some $u_i^{(k)}$, $u_{\min} < u_i^{(k)}$ or $u_i^{(k)} < u_{\max}$. Then there are two conditions based on the initial assignment vector $\mathbf{u}^{(0)}$.

1) Let the initial assignment be $u_i^{(0)} = u_{\min}$, $i \in \{1,...,Q\}$. One of two cases can occur. By Theorem 7-2 $u_i^{(k)} \leq u_i^{(k-2)}$, $i \in \{1,...,Q\}$, $k \in \{2,3,4,....\}$. Thus

$$u_i^{(k)} \leq u_i^{(k-2)} \leq \cdots \leq u_i^{(0)} \text{ for } k \text{ even}$$

$$u_i^{(k)} \leq u_i^{(k-2)} \leq \cdots \leq u_i^{(1)} \text{ for } k \text{ odd}$$

$$\leq u_i^{(0)} \text{ by (7.4)}$$

Alternately assume that $u_i^{(k)} < u_{\max}$ for some $i \in \{1,2,...,Q\}$ and $k \in \{0,1,2,...\}$ and $u_i^{(l)} \geq u_{\max}$ for $l=0,...,k-1$. By (7.1)

$$u_i^{(k)} = \frac{1}{a_{i,i}}\left[v_i - \sum_{j \neq i}^{Q} a_{i,j} u_j^{(k-1)} \right] < u_{\max}$$

or

$$v_i - \sum_{j \neq i}^{Q} a_{i,j} u_j^{(k-1)} - a_{i,i} u_{\max} < 0$$

The choice of $u_{\max}$ satisfies

$$v_i - \sum_{j \neq i}^{Q} a_{i,j} u_{\max} - u_{\max} \geq 0$$

Rearranging each yields

$$\sum_{j \neq i}^{Q} a_{i,j} u_j^{(k-1)} > v_i - a_{i,i} u_{\max}$$

$$\sum_{j \neq i}^{Q} a_{i,j} u_{\max} \leq v_i - a_{i,i} u_{\max}$$

Substituting

$$\sum_{j \neq i}^{Q} a_{i,j} u_j^{(k-1)} > \sum_{j \neq i}^{Q} a_{i,j} u_{\max}$$

Since $a_{i,j} \leq 0$ for $i,j \in \{1,2,...,Q\}$, $u_j^{(k-1)} < u_{\max}$ for at least one $u_j^{(k-1)}$, a contradiction.

2) The case for the initial choice of $u_{\max}$ is symmetric.

Theorem 7-4: Consider a sequence of $u_i^{(k)}$'s satisfying the conditions of Theorem 7-2. From the perspective of a processor P_i calculating u_i, a candidate intermediate result $u_l^{(k)}$ with a nonzero coefficient $a_{l,i}$, $l,i \in \{1,2,...,Q\}$ must satisfy the following properties

If $u_i^{(0)} = u_{\min}$, $i \in \{1,2,\cdots,Q\}$, then

$$u_l^{(k+2)} \le u_l^{(k)} + (1-\omega)(u_l^{(k)} - u_l^{(k-2)}) - \omega \frac{1}{a_{l,l}} \left[a_{l,i}(u_i^{(k+1)} - u_i^{(k-1)}) \right], \quad k \in \{0,1,2,...\}$$

Similarly if $u_i^{(0)} = u_{\max}$, $i \in \{1,2,\cdots,Q\}$, then

$$u_l^{(k+2)} \ge u_l^{(k)} + (1-\omega)(u_l^{(k)} - u_l^{(k-2)}) - \omega \frac{1}{a_{l,l}} \left[a_{l,i}(u_i^{(k+1)} - u_i^{(k-1)}) \right], \quad k \in \{0,1,2,...\}$$

Proof: Consider two successive iterations at k and $k+2$

$$u_l^{(k+2)} = (1-\omega)u_l^{(k)} - \omega \frac{1}{a_{l,l}} \left[\sum_{j \ne l, j \ne i} a_{l,j} u_j^{(k+1)} + a_{l,i} u_i^{(k+1)} - v_l \right]$$

and

$$u_l^{(k)} = (1-\omega)u_l^{(k-2)} - \omega \frac{1}{a_{l,l}} \left[\sum_{j \ne l, j \ne i} a_{l,j} u_j^{(k-1)} + a_{l,i} u_i^{(k-1)} - v_l \right]$$

Subtracting $u_l^{(k+2)} - u_l^{(k)}$

$$u_l^{(k+2)} - u_l^{(k)} = (1-\omega)(u_l^{(k)} - u_l^{(k-2)})$$

$$- \omega \frac{1}{a_{l,i}} \left[\sum_{j \ne l, j \ne i} a_{l,j} u_j^{(k+1)} + a_{l,l} u_i^{(k+1)} - \left(\sum_{j \ne l, j \ne i} a_{l,j} u_j^{(k-1)} + a_{l,i} u_i^{(k-1)} \right) \right]$$

Since $u_l^{(k-1)} \ge u_l^{(k+1)}, k \in \{1,2,...\}$

$$\le (1-\omega)(u_l^{(k)} - u_l^{(k-2)}) - \omega \frac{1}{a_{l,l}} (a_{l,i} u_i^{(k+1)} - a_{l,i} u_i^{(k-1)})$$

The case when $u_i^{(0)} = u_{max}$, $i \in \{1,2,\cdots,Q\}$ is symmetric.

Theorem 7-5: A candidate intermediate result $u_l^{(k)}$ with nonzero coefficient $a_{l,i}$, from the perspective of processor calculating $u_i^{(k)}$, $l,i \in \{1,2,...,Q\}$, must satisfy the following properties.

If $u_i^{(0)} = u_{\min}$, $i \in \{1,2,\cdots,Q\}$, then

$$u_l^{(k+2)} \geq (1-\omega)u_l^{(k)} - \omega\frac{1}{a_{l,l}}\left[\sum_{j\neq l, j\neq i} a_{l,j}u_{\max} + a_{l,i}u_i^{(k+1)} - v_l\right]$$

Similarly if $u_i^{(0)} = u_{max,}$ $i \in \{1,2,\cdots,Q\}$, then

$$u_l^{(k+2)} \leq (1-\omega)u_l^{(k)} - \omega\frac{1}{a_{l,l}}\left[\sum_{j\neq l, j\neq i} a_{l,j}u_{\min} + a_{l,i}u_i^{(k+1)} - v_l\right]$$

Proof: Immediate from the conditions of Theorem 7-2 and the proof of Theorem 7-4 □.

Theorems 7-4 and 7-5 guarantee a minimal and maximal amount of progress of each individual component of the solution as a function of the current state of the solution.

The stopping point is selected to be the iteration K at which $|u_i^{(K+2)} - u_i^{(K)}| < \varepsilon$ for $i=1,2,3,...,Q$ for some small ε.

Theorem 7-6: At the end of the final iteration K, the final result $u_i^{(K)}$ satisfies the following relations

If $u_i^{(0)} = u_{\min}$, $i \in \{1,2,\cdots,Q\}$, then

$$u_i^{(K)} \geq \frac{1}{a_{i,i}}\left[\sum_{j\neq i}^{Q} a_{i,j}u_j^{(K)} - v_i\right] \geq u_i^{(K)} - \frac{\varepsilon}{\omega}$$

and if $u_i^{(0)} = u_{\max}$, $i \in \{1,2,\cdots,Q\}$, then

$$u_i^{(K)} \leq \frac{1}{a_{i,i}}\left[\sum_{j\neq i}^{Q} a_{i,j}u_j^{(K)} - v_i\right] \leq u_i^{(K)} + \frac{\varepsilon}{\omega}$$

Proof: Consider the final iteration at $K+2$. Assume that $u_i^{(0)} = u_{\min}$, $i \in \{1,2,\cdots,Q\}$.

$$u_i^{(K+2)} = (1-\omega)u_i^{(K)} - \omega\frac{1}{a_{i,i}}\left[\sum_{j\neq i}^{Q} a_{i,j}u_j^{(K+1)} - v_i\right]$$

$$u_i^{(K+2)} - (1-\omega)u_i^{(K)} = -\omega\frac{1}{a_{i,i}}\left[\sum_{j\neq i}^{Q} a_{i,j}u_j^{(K+1)} - v_i\right]$$

Since $u_i^{(K)} - u_i^{(K+2)} < \varepsilon$,

$$\frac{\varepsilon}{\omega} > u_i^{(K)} - \frac{1}{a_{i,i}}\left[\sum_{j\neq i}^{Q} a_{i,j}u_j^{(K+1)} - v_i\right]$$

The case when $u_i^{(0)} = u_{\max}$, $i \in \{1,2,\cdots,Q\}$ is symmetric □.

Each of the theorems developed is a member of a particular subclass of constraint predicates. This membership is summarized in Table 7.1.

Subclass	Theorems	Size
Φ_P	7-2	IPT
Φ_F	7-3	IPT
Φ_C	7-4,7-5 7-6	IPT GPT

Table 7.1. Predicate Subclass Membership.

7.3 ERROR COVERAGE MODELING FOR MATRIX ITERATIVE ANALYSIS

As in the development of the constraint predicate, the development of the modeling of each individual feature is treated separately.

Feasibility

Modeling of the error coverage of the feasibility constraints for the problem considered in this chapter is straightforward. The feasibility events $F_i, i=1,2$ are defined by

${}^{j}F_1^{(k)}$: Event that $u_j^{(k)} \geq u_{\max}$ for j and k under test and

${}^{i}F_2^{(k)}$: Event that $u_i^{(k)} \leq u_{\min}$ for the i and k under test.

Since these constraints are stationary

$Pr[{}^{j}F_i] = Pr[{}^{j}F_i^{(k)}]$ for all $j \in \{1,2,...,Q\}$, $k \in \{0,1,2,...\}$, and $i = 1,2$.

The cumulative distribution function of the appropriate model yields the coverage probabilities for all steps of the problem.

The corresponding probabilities are given by

$$Pr[{}^{j}F_1] = Pr[u_j^{(k)} \geq u_{\max}]$$

$$Pr[{}^{j}F_2] = Pr[u_j^{(k)} \leq u_{\min}]$$

and

$$Pr[{}^{j}F] = Pr[u_{\max} \leq u_j^{(k)} \leq u_{\min}]$$

Progress

Progress is modeled by considering the global convergence properties of the solution and extrapolating this behavior back to each individual case. [Ames77] gives a bound on the rate of error reduction in a matrix iterative solution.

The first step in this analysis is to determine the spectral radius ρ_G of the iteration matrix **G**. It is possible to determine a close approximation to ρ_G by examining the ratio of the norm of the error vectors between iterations. However, ρ_G can also be calculated directly from **G**, while in general is not feasible due to the size of **G**, for analytical purposes is sufficient. For the red-black ordering used in this chapter, **G** cannot be expressed in a convenient matrix form. Instead we use the **G** from the Gauss-Seidel consistent ordering with Successive Over-Relaxation applied. The matrix **A** is decomposed into **A=B+D+C** with **B** lower triangular, **C** upper triangular, and **D** diagonal $Q \times Q$ matrices. Thus **G** becomes,

$$\mathbf{G} = (\mathbf{D}+\omega\mathbf{B})^{-1}[(1-\omega)\mathbf{D}-\omega\mathbf{C}]$$

If the complex eigenvalues of **G** are given by λ_i for $i \in \{1,...,Q\}$, then

$$\rho_G = \max_{i \in \{1,...,Q\}} |\lambda_i|$$

Let the error at step k of the computation be denoted by the vector $\boldsymbol{\varepsilon}^{(k)} = |\mathbf{u}^k - \mathbf{u}|$ for $k \in \{0,1,2,...\}$. For a stationary linear iteration matrix **G** of the form considered in Theorem 7-1, $\|\boldsymbol{\varepsilon}^{(k)}\| = \|\mathbf{G}^k \boldsymbol{\varepsilon}^{(0)}\|$ where $\|\boldsymbol{\varepsilon}^{(k)}\|$ is the spectral norm of $\boldsymbol{\varepsilon}^{(k)}$. If $\rho_G < 1$, for k sufficiently large, $\|\mathbf{G}^k\| \approx [\rho_G]^k$ where $\|\mathbf{G}\|$ is the spectral norm of **G**. Combining these results yields [Ames77]

$$\|\boldsymbol{\varepsilon}^{(k)}\| \leq \|\mathbf{G}^k\| \, \|\boldsymbol{\varepsilon}^{(0)}\|$$

Thus, for large k the ratio $\|\boldsymbol{\varepsilon}^{(k+1)}\| / \|\boldsymbol{\varepsilon}^{(k)}\|$ averages to ρ_G. Therefore, on the average, the error decreases by a factor of ρ_G at each step in the iteration. For modeling purposes, the final result is known. Assuming that each individual point behaves according to global convergence and using Theorem 7-2, treatment of an arbitrary point, i', yields the following relation

If $u_i^{(0)} = u_{\min}$,

$$\rho_G^k (u_{\min} - u_{i'}) \geq u_i^{(k)} - u_{i'}, \; k \in \{0,1,2,...\} \tag{7.5}$$

where $u_{i'}$ is the correct final result of the calculation. A similar result holds if $u_i^{(0)} = u_{\max}$. The events P_i that define progress are

If $u_i^{(0)} = u_{\min}$,

$^{i'}P_1^{(k)}$ is the event $u_i^{(k)} \leq \rho_G^k (u_{\min} - u_{i'}) + u_{i'}, \; k \in \{0,1,2,...\}$

If $u_i^{(0)} = u_{\max}$,

$^{i'}P_2^{(k)}$ is the event $u_i^{(k)} \geq u_{i'} - \rho_G^k (u_{i'} - u_{\max}), \; k \in \{0,1,2,...\}$

Consistency

The consistency components of Theorems 7-4 and 7-5, like the progress component, can either be determined from a trace or by analytic means. For the analytic model, we apply Theorem 7-4 with the assumption that each u_i proceeds toward the solution U at

the same convergence rate. Again this is not the case in the actual solution but does effectively display the aggregate behavior of the solution. While Theorems 7-4 and 7-5 are directly implementable as a constraint predicate, to use them in the model requires additional simplification. The ratio $a_{l,i}/a_{l,l}$ is aggregated by replacing it with the ratio $R=\frac{u_i^{(k)}}{nu_i^{(k-1)}}$ where n is the average number of nonzero $a_{l,i}$'s. Using (7.5) as an equality and substituting into Theorem 7-4 yields the events ${}^{i'}C_{1a}^{(k+2)}$ and ${}^{i'}C_{1b}^{(k+2)}$

If $u_{i'}^{(0)}=u_{\min}$, ${}^{i'}C_{1a}^{(k+2)}$ is the event,

$$u_{i'}^{(k+2)} \le \rho_G^k(u_{\min}-u_{i'})-u_{i'}(1-\omega)(\rho_G^k(u_{\min}-u_{i'})-\rho_G^{k-2}(u_{\min}-u_{i'})$$

$$-\omega R(\rho_G^{k+1}(u_{\min}-u_{i'})-\rho_G^{k-1}(u_{\min}-u_{i'})),\ k\in\{2,3,4,...\};$$

and if $u_{i'}^{(0)}=u_{\max}$, ${}^{i'}C_{1b}^{(k+2)}$ is the event,

$$u_{i'}^{(k+2)} \ge \rho_G^k(u_{\min}-u_{i'})-u_{i'}(1-\omega)(\rho_G^k(u_{\min}-u_{i'})-\rho_G^{k-2}(u_{\min}-u_{i'})$$

$$-\omega R(\rho_G^{k+1}(u_{\min}-u_{i'})-\rho_G^{k-1}(u_{\min}-u_{i'})),\ k\in\{2,3,4,...\}.$$

Letting the mean of the v_i's be $\bar{v}$, substitution into Theorem 7-5 yields the events ${}^{i'}C_{2a}^{(k+2)}$ and ${}^{i'}C_{2b}^{(k+2)}$.

If $u_{i'}^{(0)}=u_{\min}$, ${}^{i'}C_{2a}^{(k+2)}$ is the event,

$$u_{i'}^{(k+2)} \ge (1-\omega)(\rho_G^k(u_{\min}-u_{i'})+u_{i'}$$

$$-\omega R\left[\sum_{j\neq l,j\neq i}^{Q} u_{\max}+\rho_G^{k+1}(u_{\min}-u_{i'})+u_{i'}-\bar{v}\right],\ k\in\{2,3,4,...\};$$

and if $u_{i'}^{(0)}=u_{\max}$, ${}^{i'}C_{2a}^{(k+2)}$ is the event,

$$u_{i'}^{(k+2)} \le (1-\omega)(u_{i'}-\rho_G^k(u_{i'}-u_{\max})$$

$$-\omega R\left[\sum_{j\neq l,j\neq i}^{Q} u_{\min}+u_{i'}-\rho_G^{k+1}(u_{i'}-u_{\min})-\bar{v}\right],\ k\in\{2,3,4,...\}.$$

With these relationships defined, the modeling can proceed. Choose as an example problem the matrix shown in Figure 7.2 which satisfies the conditions of Theorem 7-1.

The spectral radius ρ_G of the corresponding iteration matrix G is 0.4775 for ω=1. It is easily verified that u_{min}=1.00 (from rows 5 or 6 of A) and u_{max}=.057 (from row 4 of A)†.

$$\mathbf{A} = \begin{bmatrix} 4.87 & -1.28 & 0 & -1.37 & 0 & 0 & 0 & 0 & 0 \\ -1.28 & 5.62 & -1.78 & 0 & -1.37 & 0 & 0 & 0 & 0 \\ 0 & -1.78 & 6.87 & 0 & 0 & -1.37 & 0 & 0 & 0 \\ -1.37 & 0 & 0 & 5.37 & -1.28 & 0 & -1.62 & 0 & 0 \\ 0 & -1.37 & 0 & -1.28 & 6.12 & -1.78 & 0 & -1.62 & 0 \\ 0 & 0 & -1.37 & 0 & -1.78 & 7.37 & 0 & 0 & -1.62 \\ 0 & 0 & 0 & -1.62 & 0 & 0 & 5.87 & -1.28 & 0 \\ 0 & 0 & 0 & 0 & -1.62 & 0 & -1.28 & 6.62 & -1.78 \\ 0 & 0 & 0 & 0 & 0 & -1.62 & 0 & -1.78 & 7.87 \end{bmatrix}$$

$$\mathbf{v} = \begin{bmatrix} 0.343750 \\ 0.625000 \\ 3.437500 \\ 0.062500 \\ 0.062500 \\ 2.593750 \\ 0.179688 \\ 0.531250 \\ 3.648438 \end{bmatrix}$$

Figure 7.2. Sample Matrix for **Au=v**

Matrix formed from the finite difference solution of the self-adjoint elliptic PDE $\frac{\partial}{\partial x}(x+1)\partial\frac{u}{\partial x}+\frac{\partial}{\partial y}(y^2+1)\partial\frac{u}{\partial y}-u=1$ on the unit square with Dirichlet Boundary conditions:

$u(0,y)=y$, $u(1,y)=y^2$, $u(x,0)=0$, $u(x,1)=1$.

The normal distribution function is chosen as the conditional distribution of erroring values. If we center the mean of the distribution at the actual correct result of the calculation, then the normal with mean μ and standard deviation σ is an attractive choice since the erroring values tend to group near the correct value. While an overly pessimistic assumption, it does display the effectiveness of the generated constraint predicate.

†Note that due to truncation of the values shown these values do not correspond exactly with the matrix shown.

Figure 7.3 depicts the feasibility event F and its relation to the conditional distribution of erroring values. $\mu = .562667$ is equal to the norm of the final solution vector $\| \mathbf{u}^{(k)} \| / Q$. $\sigma = .65$ is assigned such that more than half of the error probability is undetectable by the feasibility conditions.

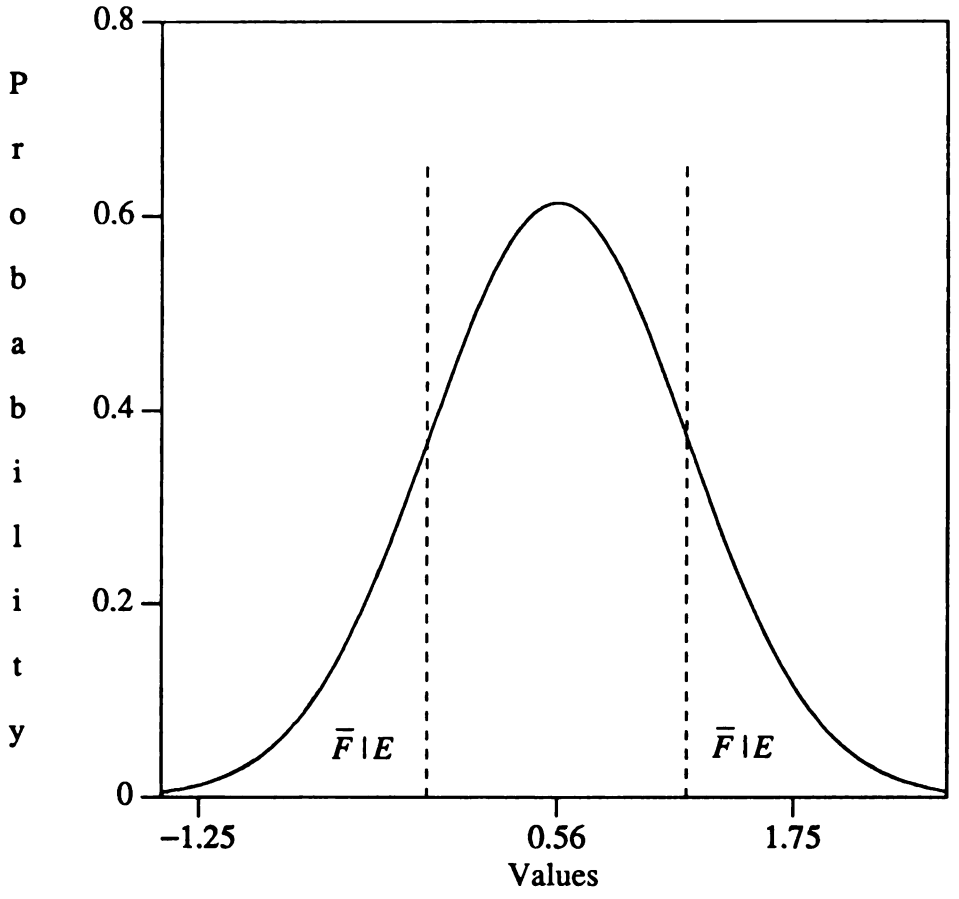

Figure 7.3. Feasibility Event and
Conditional Distribution of Errors

Let E be the event that an error has occurred. Figure 7.4 depicts the error coverage as a function of the iteration step k for three different levels of error coverage. The error coverage of the feasibility predicate is constant since the constraints are stationary. The progress component rapidly rises to it's limiting value of approximately 25% additional

coverage. This is because the average solution makes the greatest reduction of error in the early steps of the solution. The consistency component adds only a few more percent to the error coverage to bring the total error coverage to approximately 75%.

Seventy-five percent of the errors generated by a faulty processor can be flagged with a fault latency of 1 using only IPT predicates. It is interesting to note that while the addition of the consistency constraints only provides a small gain in error coverage, it is still a necessary component. Constraints C_{1a} and C_{1b} guarantee that a faulty processor must make some non-negligible movement toward the solution. While not modeled here, empirical data suggest that a faulty processor can delay convergence almost indefinitely by chosing a movement just slightly large than ε. The consistency constraint, when viewed in this manner, is a necessary component of the constraint predicate.

The choice of the normal as the distribution of errors also limits the error coverage using these predicates. Any symmetric distribution centered at the final result will necessarily result in an error coverage measurement somewhat less that total coverage since the tests developed (except for the consistency test based on Theorem 7-5) are primarily one-sided.

Theorem 7-6, the GPT predicate, has not yet been used in the constraint predicate. Theorem 7-6 affords total test coverage if certain additional restrictions are imposed.

As mentioned previously, algorithm *Agree* performs a reliable broadcast such that all recipients of a message receive the same version of that message and the message received is the message sent. Employing *Agree* requires a bound on the number of failures that may occur. To utilize Theorem 7-6, the number of failures must be bound as well from the application level.

Definition 7-3: Local Fault Group: $\Omega_i = \{P_i | a_{i,j} \neq 0,\ j=1,...,N\}$.

Thus a fault group $\Omega_{i'}$ for some i' is the set of processors involved directly in the solution of (7.1) for point $u_{i'}$.

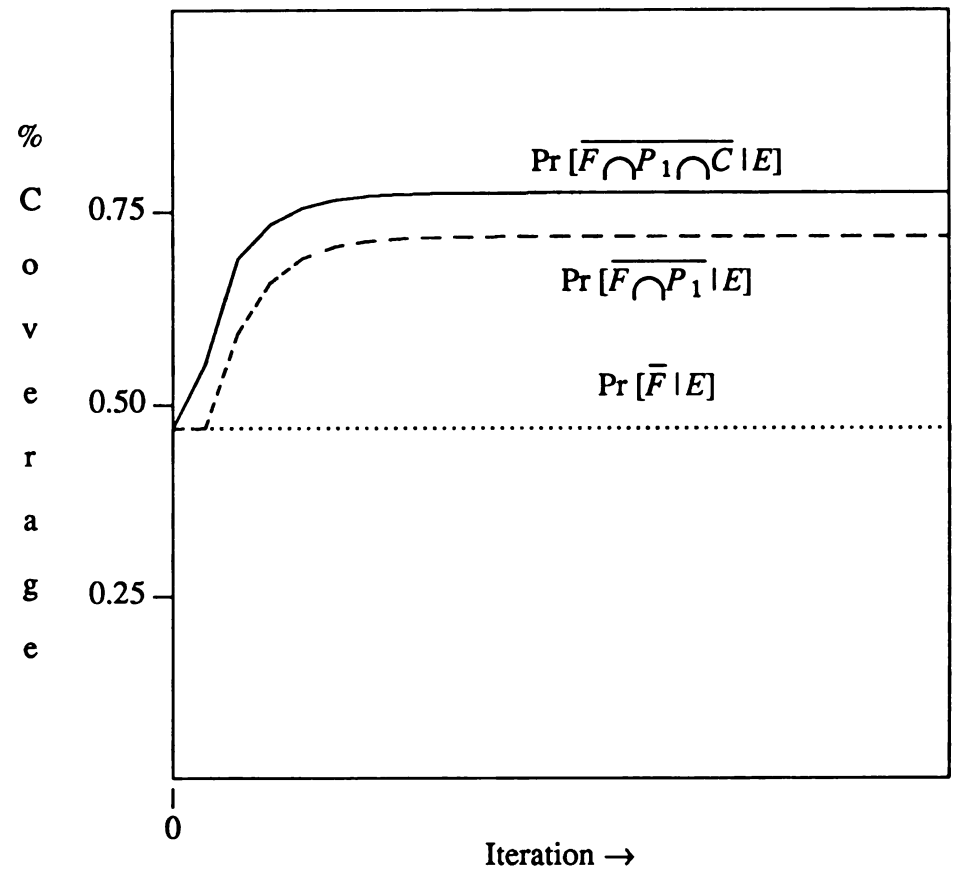

Figure 7.4. Error Coverage by Solution Step Count

The diagnosis of a fault group is stated as follows. If a processor in $\Omega_{i'}$ is faulty for some i', then $\Omega_{i'}^{D} \equiv 1$ and the entire fault group is flagged as faulty; otherwise, $\Omega_{i'}^{D} \equiv 0$ and the entire fault group is flagged as non-faulty.

Lemma 7-1: Let the processors in a local fault group $\Omega_{i'}$ communicate via *Agree*. Then processors in $\Omega_{i'}$ reliably diagnose $\Omega_{i'}$ as either faulty or non-faulty if the maximum number of faulty units per fault group is 1.

Proof: The proof is constructive.

Step 1: Each Processor $P_j \in \Omega_{i'}$ sends its last value of $u_j^{(K)}$ to the other members of $\Omega_{i'}$.

Step 2: If $|u_i^{(K-2)} - u_i^{(K)}| > \varepsilon$ then report that $\Omega_{i'}$ (and indeed $P_{i'}$) is faulty. Stop. Otherwise,

Step 3: Each processor applies Theorem 7-6 to the $u_j^{(K)}$'s it receives from Step 1.

Step 4: Each processor reports, in a distributed manner, whether Theorem 7-6 is satisfied or not.

Since a reliable broadcast is utilized, each processor receives the same version of a sent message. Each non-faulty processor then reaches the same conclusion concerning the result of the final calculation of the iteration sequence specified by (7.1). The test in Step 2 precludes $P_{i'}$ changing its value from an erroneous to correct state during the broadcast phase. If $P_{i'}$ were allowed to change it's value, it could send a value that agreed with Theorem 7-6 thus making $\Omega_{i'}^D = 0$ erroneously.

If $P_{i'}$ does not change it's value, then Theorem 7-6 is used to diagnose $\Omega_{i'}$. Note that Theorem 7-6 does not explicitly specify which member of $\Omega_{i'}$ is faulty. Since the maximum number of faults per fault group is 1, no two processors can cooperate to fool the test of Theorem 7-6. Furthermore, in the example under consideration, since the minimum cardinality fault group is 3, the non-faulty processors will always outvote the faulty processors. A simple majority vote is taken to determine the status of the fault group. □

Definition 7-4: Extended Fault Region $K_i = \bigcup\{\Omega_j | \Omega_j \cap P_i \neq \varnothing\}$

The extended fault region of a point i' is simply the set of local fault groups of all neighboring points j such that u_j uses $u_{i'}$ in it's calculation.

Let $K_i^D = |\{P_j | P_j \text{ is faulty and } P_j \in K_i\}|$ and let $k_i = |\{j | \Omega_j \in K_i\}|$

Theorem 7-7: For each processor P_i, P_i is faulty iff $\Omega_j^D = 1$ for all $\Omega_j \in K_i$ and $K_i^D \leq k_i - 2$.

Proof:

Necessity: Assume P_i is nonfaulty. If $\Omega_i^D = 0$ then we are done. If not, then there exists some $P_j \in \Omega_i$ and $P_j \in \Omega_j$ such that P_j is faulty. Since at most $k_i - 2$ processors may be

faulty and at most one is allowed per fault group (by Lemma 7-1), and since there are k_i-2-1 faulty processors remaining to be distributed over the k_i-2 remaining fault groups, some $\Omega_i^D = 0$.

Sufficiency: Assume that some $\Omega_j^D = 0$. However, by Definition 7-4, each $\Omega_j \in K_i$ contains P_j. Thus P_j is nonfaulty. Secondly assume that $K_i^D > k_i-2$. Then it is possible to have k_i-2 faulty processors none of which are in Ω_i and one faulty processor in Ω_i which is not P_i. Thus P_i is nonfaulty. □

With the test provided by Theorem 7-7, we can obtain the error coverage shown in Figure 7.5.

7.4 RUN TIME PERFORMANCE

Using the techniques of Chapter 6 for this problem will yield a similar performance curve to that of Figure 6.1 (p. 80). However, in matrix iterative solutions, many points will be assigned to a single processor. All of the constraint predicates developed in this chapter are applicable, but they only need to be run at those points which fall at the boundary of the data partitioning. Let the total number of points on the grid Q be greater than the number of processors N. Without loss of generality assume that N divides Q into a perfect square. Then $Q/N = q$ is the number of points per processor. Continuing the assumption of coordinate grid indexed points, Figure 7.6 depicts the mapping and error checking for $q = 25$. The grids marked by dashed lines indicate points that are not checked by the constraint predicate as they are internal to that processor.

The computational complexity calculation proceeds along the lines of Chapter 6. The unreliable algorithm, for a sub-grid of q elements per processor with an average of $n-1$ neighbors, is easily seen to have a computational complexity per iteration given by,

$$C_{pNR} = nq + 2S_L.$$

Each constraint predicate contributes the following run time overhead,

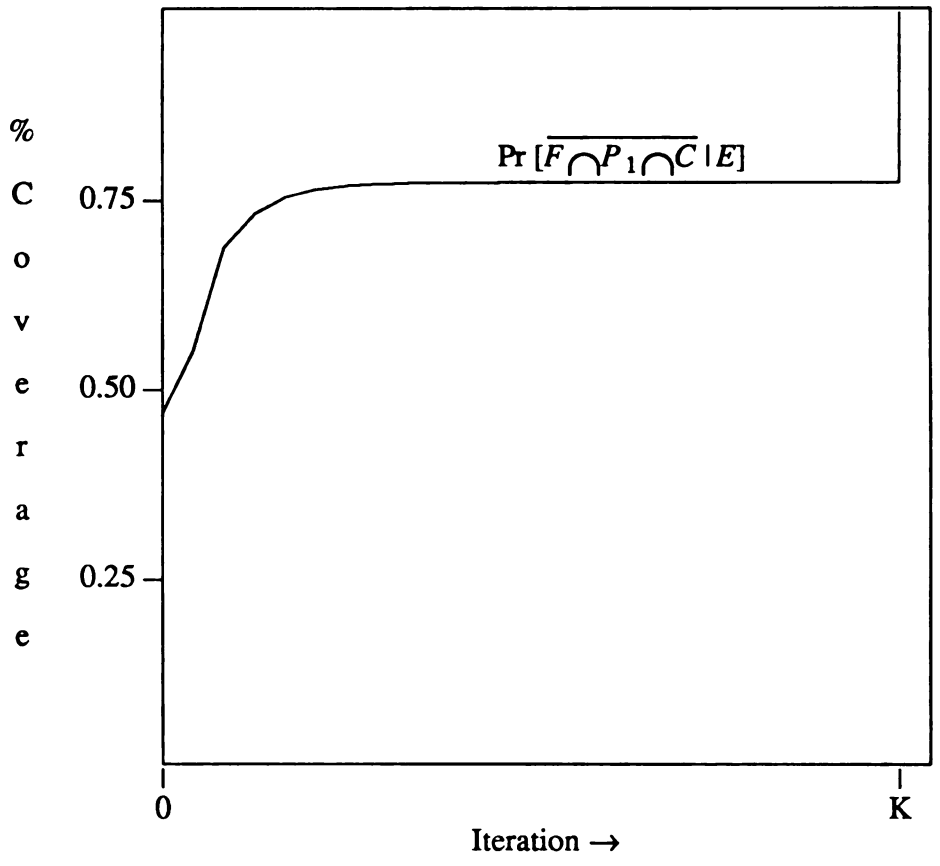

Figure 7.5. Error Coverage by Solution Step Count.

Theorem 7-7 Applied.

Φ_P 1

Φ_F $n\sqrt{q}$

Φ_C $1+n\sqrt{q}$

The run time of the reliable algorithm under a maximum of t faults per fault group is given by

$$C_{pR} = 2+2n\sqrt{q}+nq+2(t+1)S_L+(n\sqrt{q})^{t+1}$$

The ratio of the computation times of the reliable algorithm to the unreliable algorithm is:

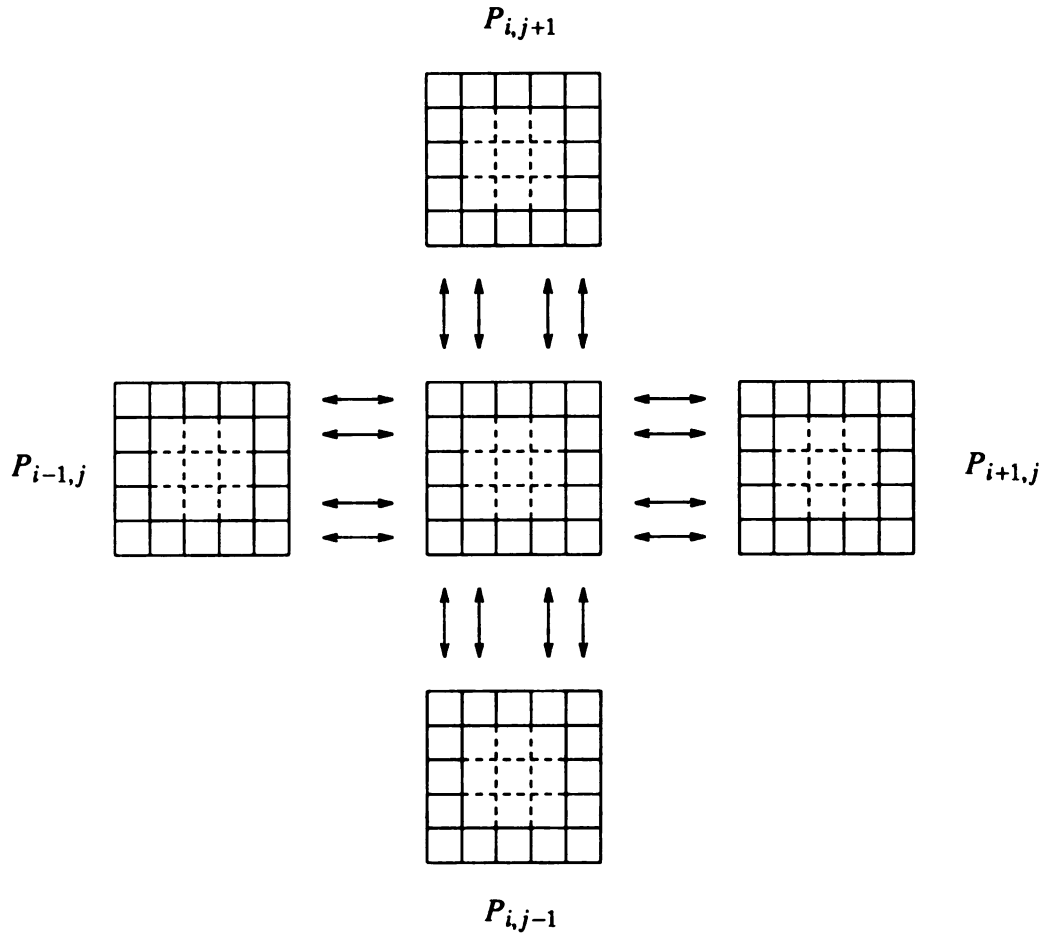

Figure 7.6. Mapping for q=25.

$$\frac{C_{pR}}{C_{pNR}} = \left[\frac{1+2n\sqrt{q}+nq+2(t+1)S_L+(n\sqrt{q})^{t+1}}{nq+2S_L} \right]$$

Let r represent the computation to communication ratio. Then $r = nq/2S_L$. Neglecting low order terms, for t=1 (the solution is locally tolerant of one fault per fault group Ω_i),

$$\frac{C_{pR}}{C_{pNR}} = \frac{2rS_L+4S_L+2rS_L}{2S_L(r+1)}$$

$$= \frac{5r+2}{r+1} \qquad (7.6)$$

By setting r=0, (7.6) is easily seen to be a more general form of (6.9). Letting r=1 sets computation time equal to communication time resulting in,

$$C_R \approx \frac{n+2}{2} C_{NR}. \tag{7.7}$$

Since n is usually very small (5 or 7), the reliable algorithm is at most 3.5 to 4.5 times more expensive than the unreliable algorithm and indeed much smaller than this since the value for q to achieve even an $r=1$ is prohibitively large. Moreover, this analysis is overly pessimistic as it treats logical comparisons with the same weight as floating point multiplication/division.

7.5 COMMENTS AND LIMITATIONS

In SSOR, the optimal relaxation factor ω_{opt} is chosen to minimize the spectral radius of the iteration matrix **G**. However, setting $\omega = \omega_{opt}$, does not preserve the monotonicity of the u_i's as was suggested in [HuAb84]. The expectation of monotonic behavior is unreasonable since SOR is specifically designed to "over-relax" some u_i's to increase the convergence rate. Values of ω in the range $1 < \omega \leq \omega_{opt}$ can be found experimentally that do preserve monotonicity; however, no way is known to calculate these values even if the spectral radius is known a priori. This limits the feasibility of using constraint predicates to iteration matrices that have the spectral radius $\rho_G \ll 1$. It is not known at this time whether a convergence envelope can be found to bound the u_i's. This is an area for further study.

Error coverage can detect a significant number of errors with a fault-latency of 1. However, some errors can result in a pathological state of the system (values of $\mathbf{u}^{(k)}$ for some k in which convergence to the final correct result takes longer than the time required to restart the solution from the initial guess). Enhanced error checking capability can come only from an increased understanding of the local convergence properties of the solution. Such work is being performed under the guise of "Local Relaxation Methods" [KuLM87, BoVe82]. Local convergence properties, as shown in this chapter, can be embodied as IPT predicates.

A final comment on the constraint predicate implementation concerns Theorem 7-7. One of the goals of application oriented testing is to migrate the system dependent issue to the diagnostic basis. Theorem 7-7 is an Ad Hoc approach, at best, to fault diagnosis. Optimally, control of a GPT test such as this should be migrated to the diagnostic basis. It is not understood at this point in time how to do this in a general way while still preserving the application oriented view of fault-tolerance. Clearly this is an area for further study.

Chapter 8

Relaxation Labeling

We now focus our attention on a non-numerical problem, Relaxation Labeling. To describe the problem of relaxation labeling, we quote from [HuZu83]:

> *Relaxation labeling processes are a class of mechanisms that were originally developed to deal with ambiguity and noise in vision systems. The general framework, however, has far broader potential applications and implications. The structure of relaxation labeling is motivated by two basic concerns: 1) the decomposition of a complex computation into a network of simple "myopic," or local, computations; and 2) the requisite use of context in resolving ambiguities...*

Fundamentally, relaxation labeling assigns labels to objects in an image. Various compatibilities and incompatibilities exist between the objects. For example, consider an image which is to be labeled as a human body. If two objects are adjacent in the image and one is the object "head," then it is very likely that an adjacent object is "shoulder." Likewise, it is highly improbable that the adjacent object has the label "foot." This competition and cooperation can work together in the relaxing the labelings on the objects such that eventually an unambiguous labeling is found.

Parallelization of relaxation labeling for implementation on a multiprocessor system is advantageous for large problems, that is, problems that would otherwise take a long time on a conventional SISD computer. A large problem is characterized by a large number of objects. Indeed some applications of relaxation labeling can include objects the size of pixels in a 512x512 image or 2^{18} objects [Stoc87]. Relaxation labeling lends itself easily to implementation as a data parallel algorithm [HiSt86]. The algorithm

requires only local computations and data exchanges. Thus good speedup may be obtained by parallelization. As in the matrix iterative case, the properties of the intermediate results of the computation are checked for faulty/nonfaulty behavior.

8.1 PARALLELIZED WEIGHTED RELAXATION LABELING

The weighted relaxation labeling algorithm using the variational inequality method given by [HuZu83] yields a straightforward parallelization. The notation and algorithms are presented here for completeness. The basic idea is that given an initial feasible solution, attempt to maximize an objective function by taking small steps in the tangent direction which maximizes the directional derivative of the objective function. When the directional derivative becomes negative or zero, the objective function is at a local maximum and the procedure stops. N is the number of objects and m is the number of possible labels. Labels are denoted by λ and λ'. An objective function is defined as:

$$q_i(\lambda)=\sum_{\lambda=1}^{m} p_i(\lambda)s_i(\lambda), \quad 1 \le i \le N \tag{8.1}$$

where the support function $s_i(\lambda)$ is:

$$s_i(\lambda)=\sum_{j=1}^{N}\sum_{\lambda'=1}^{m} r_{ij}(\lambda,\lambda')p_j(\lambda') \tag{8.2}$$

The compatibility matrix $R_{N\times N} = [r_{ij}(\lambda,\lambda')]$ is the relative compatibility of label λ at object i with label λ' at object j.

The weight vector $\vec{p_i}$ with components $p_i(\lambda)$ is the relative weighting for label λ at object i and is constrained by:

$$\sum_{\lambda=1}^{m} p_i(\lambda)=1, \quad 0\le p_i(\lambda)\le 1, \quad 1\le \lambda \le m \tag{8.3}$$

The weight vector $\vec{p_i}$ is constrained to be a consistent solution for each object i providing:

$$\sum_{\lambda=1}^{m} p_i(\lambda)s_i(\lambda) \ge \sum_{\lambda=1}^{m} v_i(\lambda)s_i(\lambda) \text{ for all labelings } \vec{v_i} \text{ satisfying (8.3)} \tag{8.4}$$

8.1.1 Non-Fault-Tolerant Parallel Relaxation Algorithm

Parallelization of the algorithm is accomplished through assignment of each of the N objects to one of the N processors in the DMMP. The i'th processor is denoted by P_i. Each processor runs the same copy of the algorithm on different data, hence the term data parallel algorithm. Each processor P_i has a copy of the entries relevant to it of the compatibility coefficient matrix $R_{N\times N}$.

The *neighbors* of a processor in the problem domain of relaxation labeling are those that have non-zero compatibility coefficients. Thus each processor P_i has only the nonzero column elements j of row i in $R_{N\times N}$. The average number of nonzero entries per row is given by n.

In the following algorithm, the index i indicates the local object (processor), and the index j ranges over the neighbors. For example, $\vec{p}_j^{\,k}$ indicates the k'th iteration value of $\vec{p}$ held by processor P_j. Each processor P_i executes the following algorithm $1 \le i \le N$. It is assumed here that the mapping is ideal. That is, each processor has a direct connection to its neighbors. This assumption is not critical and it simplifies complexity analysis.

Initialize:
1) Start with a consistent initial labeling assignment $\vec{p}^{0}$
$k \leftarrow 0$
loop
2a) **send** $\vec{p}_i^{k}$ to all n neighbors P_j
2b) **receive** $\vec{p}_j^{k}$ from all n neighbors P_j

/* nearest neighbor updating */
2c) $s_i(\lambda) = \sum_{j}^{n} \sum_{\lambda'}^{m} r_{ij}(\lambda,\lambda')p_j(\lambda')$

/* find a feasible gradient direction */
3) $\vec{u}^{k} = projection\text{-}operator(\vec{p}^{k}, \vec{s}^{k})$

4) **if** ($\vec{u}^{k}$=0) **break;**

5) $\vec{p}^{k+1} \leftarrow \vec{p}^{k} + h\vec{u}^{k}$
/* where h is some small positive value */
/* that keeps $\vec{p}^{k+1}$ consistent

6) $k \leftarrow k+1$
pool

Figure 8.1. Algorithm RL_{NR},

Relaxation labeling with no added reliability

8.1.1.1 Projection Operator Computation

Computation of the updating direction vector is presented in [MoHZ83]. It is formulated as a linear optimization problem subject to quadratic constraints. We repeat the formal problem specification here.

Let IR^m be m-dimensional real space and let IK be the convex set defined by:

$$IK = \left\{ \vec{p} \in IR^m \mid \sum_{\lambda=1}^{m} p(\lambda)=1,\ p(\lambda) \geq 0,\ 1 \leq \lambda \leq m \right\}$$

For any vector $\vec{p} \in IK$, the tangent set $T_{\vec{p}}$ is:

$$T_{\vec{p}} = \left\{ \vec{v} \in IR^m \mid \sum_{\lambda=1}^{m} v(\lambda)=0,\ v(\lambda) \geq 0 \ whenever\ p(\lambda)=0 \right\}$$

The set of feasible directions to move at point $\vec{p}$ is:

$$F_{\vec{p}} = T_{\vec{p}} \cap \left\{ \vec{v} \in IR^m \mid \|\vec{v}\| \leq 1 \right\}$$

The subproblem to be solved by the projection operator algorithm is; Given a current feasible weighting vector $\vec{p}$ and a current arbitrary direction $\vec{s} \in IR^m$, find $\vec{u} \in F_{\vec{p}}$ such that $\vec{s} \cdot \vec{u} \geq \vec{s} \cdot \vec{v}$ for all $\vec{v} \in F_{\vec{p}}$.

---

projection-operator $(\vec{p}, \vec{s})$

1) $D \leftarrow \{\lambda \mid p(\lambda)=0\}$
2) $S_k \leftarrow \{\}$

loop

3) $t_k \leftarrow \frac{i}{n-|S_k|} \sum_{\lambda \in S_k} s(\lambda)$;

4) $S_{k+1} \leftarrow \left\{ \lambda \in D \mid s(\lambda) < t_k \right\}$;

5) **if** $(S_{k+1}=S_k)$ **break**;
6) $k \leftarrow k+1$

pool

7) $u(\lambda) \leftarrow \begin{cases} 0 & \text{if } \lambda \in S_k \\ s(\lambda) - t_k & \text{otherwise} \end{cases}$

8) $\vec{u} \leftarrow \begin{cases} 0 & \text{if } \vec{u} = 0 \\ \dfrac{\vec{u}}{\|\vec{u}\|} & \text{otherwise} \end{cases}$

Figure 8.2. Algorithm *projection-operator*

Note here that the vector $\vec{u}$ is normalized only locally. Since it is stated in the original paper that normalization over the entire problem is not a requirement for convergence, the local normalization makes the parallel algorithm possible.

8.1.2 PERFORMANCE EVALUATION

Speedup can be defined as the ratio of the sequential complexity vs. parallel complexity. For each processor, P_i, the complexity of the *projection-operator* routine is $O(m^2)$. All other steps for one iteration are $O(m)$ with the exception of the calculation of $\vec{s_i}$ which is $O(nm)$. Thus the complexity of a parallel iteration is $O(nm+m^2+2S_L)$. If a single processor works on all the calculations, then an iteration is $O(nm+m^2)$. Thus the speedup obtain by parallelization is the ratio:

$$O\left[\frac{N(m^2+nm)}{m^2+nm+2S_L}\right]$$

If N is large, the speedup is significant.

8.2 CONSTRAINT PREDICATE

The mathematical theory surrounding the application must be mature to facilitate development of an appropriate testing predicate. Such is the case with the relaxation labeling procedure given by [HuZu83] which is essentially a constrained local maximization problem with the direction of updating solved by the method of feasible directions [Zout76].

We prove a series of lemmas which correspond to the components of the constraint predicate.

Lemma 8-1: At each iteration, the following feasibility conditions hold Φ_F:

$$\sum_{\lambda=1}^{m} p_i(\lambda) = 1 \text{ and } 0 \le p_i(\lambda) \le 1, \quad 1 \le \lambda \le m$$

Proof: Since Step 5 of the algorithm RL_{NR} in Figure 8.1 chooses h such that $\vec{p}^{\,k+1}$ remains consistent, each $\vec{p}^{\,k+1}$ must satisfy the consistency conditions. Since the consistency constraints form a compact convex set over IR^m, then all intermediate solutions must remain within a convex cone in m-space [Zout76]. This "funnels" the solution to the local attractor. □

Lemma 8-2: At each iteration the following additional feasibility conditions Φ_F are maintained:

a) $\sum_{\lambda=1}^{m} u_i(\lambda)=0$

b) $(\vec{s}-\vec{u})\cdot\vec{u}=0$

Proof:

a) Step 3 of *projection-operator*, performs an orthogonal projection of $\vec{s}$ onto the feasibility space $F_{\vec{p}}$. Thus $(\vec{s}-\vec{u})\cdot\vec{u}=0$.

b) Since $\vec{u}$ lies in $T_{\vec{p}}$, $\sum_{\lambda=1}^{m} u_i(\lambda)=0$ for all feasible $\vec{u}$. □

Lemma 8-3: Each iteration is subject to the following progress conditions Φ_P:

a) If $\vec{u} \in F_{\vec{p}}$ and $\|\vec{u}\|=1$, then $\vec{u}$ maximizes the gradient direction change.

b) If $\vec{u}$ is a correct result of projection-operator and $\vec{u}\neq 0$, then $\vec{s}_i^{k+1}\cdot\vec{p}^{k+1} > \vec{s}_i^{k}\cdot\vec{p}^{k}$

c) Convergence occurs in a finite time within a suitable neighborhood of the solution under conditions of strict consistency.

Proof:

a) Immediate from the problem definition of projection-operator.

b) If $\vec{s}_i^{k+1}\cdot\vec{p}^{k+1} < \vec{s}_i^{k}\cdot\vec{p}^{k}$ then for some $\vec{v} \in F_{\vec{p}}$, $\vec{v}\cdot\vec{s} > \vec{u}\cdot\vec{s}$. Thus $\vec{u}$ was not a correct result from projection-operator.

If $\vec{s}_i^{k+1}\cdot\vec{p}^{k+1} = \vec{s}_i^{k}\cdot\vec{p}^{k}$ then by Step 5 of RL_{NR}, since $h>0$, $\vec{u}=0$.

c) Immediate from Theorem 9.1 [HuZu83].

8.2.1 Predicate Generation and Proof

The correctness and completeness of the predicate is proven in the following theorem. The metric for correctness requires that if the algorithm RL_{NR}, under no faults, produces a correct solution, then the reliable algorithm RL_R , under a locally bounded number of faults t, also produces the same correct solution within some predetermined

error tolerance ε_0.

It will be shown that the constraints given in Lemmas 1-3 are sufficient for the predicate correctness if one constraint is added. This constraint has nothing to do with the problem theory. It is possible for a Byzantine processor to arrange the vectors $\vec{u}$ and $\vec{p}$ as to erroneously signal early stopping. Thus in the case of early stopping, the last calculation must be replicated. If the replication also produces a stopping point, then that solution is indeed the correct solution.

The method of proof is by considering all possible movements of the vector $\vec{p}$ during the iterations of the labeling algorithm.

Theorem 8-1: A predicate Φ that embodies the features Φ_P and Φ_F proven in Lemmas 1-3 plus the early stopping test will constrain Byzantine behavior such that if the unreliable solution computed be RL_{NR} is correct, the reliable solution with predicate Φ is also correct within the error tolerance ε_0.

Proof: Consider a point $\vec{p}^{\,k}$ that satisfies Lemma 8-1. A direction of movement by $\vec{u}$ may be to any point in m-space $\vec{p}^{\,k+1}$. Consider a movement by processor P_j (object j) that does not change $\vec{p}_j^{\,k}$, such that $\vec{s}_j^{\,k+1}$ is moved towards the local maximum, and $\vec{p}_j$ has not already been found to be a solution. There are two cases:

1) Lemma 8-3b is violated (Figure 8.3). If $\vec{u}_j^{\,k+1} \neq 0$, then processor P_j has made a "nonconvergence error." If $\vec{u}_j^{\,k+1} = 0$, then we must invoke the early stopping test. The members of the local fault group "re-run" the last iteration based on the values $\vec{p}_j^{\,k}$ and $\vec{u}_j^{\,k}$. If the replicated calculation is the same as the original $\vec{p}^{\,k+1}$, $\vec{u}^{\,k+1}$, then $\vec{p}_j^{\,k+1}$ is a solution and processor P_j stops.
2) $\vec{p}_j$ satisfies Lemma 8-3 but does not satisfy Lemma 8-1. Thus $\vec{p}_j$ has moved "outside" of the solution space (Figure 8.4) and violated the consistency conditions.

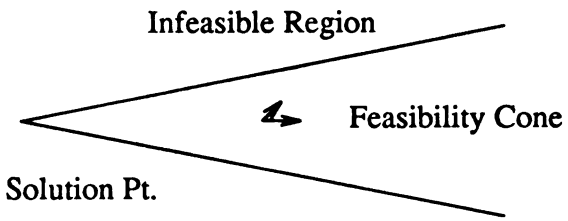

Figure 8.3. Non-maximizing movements

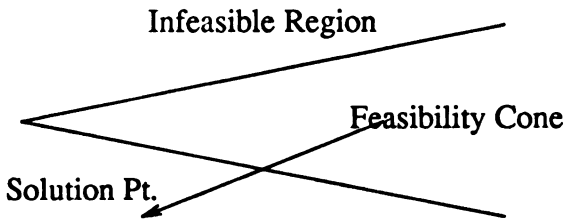

Figure 8.4. Movement outside of feasibility cone

If $\vec{p}_j^{k+1}$, $\vec{u}^{k+1}$ leads to a correct movement of $\vec{s}_j^{k}$ but does not make the maximal movement available, then Lemma 8-2 and Lemma 8-3a are violated.

Finally if Lemmas 8-1, 8-2 are satisfied and conditions of Lemma 8-3a and 8-3b are satisfied and condition 3c is violated, then the Byzantine resource is attempting to delay convergence through zigzagging (Figure 8.5). However, zigzagging may occur as a result of errors even if the algorithm employs an anti-zigzagging procedure. Even though the solution is making progress by virtue of condition (a), for all practical purposes it could be making progress as small as a 1 bit change in $\vec{s}_j$. Clearly this would cause the solution to run a very long time. However, we know that the convergence occurs within a finite amount of time, so the following test is utilized:

If $\vec{s}_i^{k+1} \cdot \vec{p}^{k+1} - \vec{s}_i^{k} \cdot \vec{p}^{k} < \varepsilon^{k+1}$ then a convergence error is flagged. When $\varepsilon^k < \varepsilon_0$ for the predetermined error tolerance ε_0, the solution halts.

Thus the predicate Φ catches all movement errors that will lead to incorrect and slow solutions. □

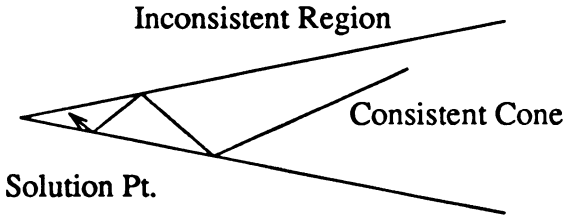

Figure 8.5. Zigzagging - Convergence but no Finiteness

8.3 RELIABLE PARALLEL ALGORITHM FOR RELAXATION LABELING

The reliable algorithm is constructed from the proof of Theorem 8-1. At each iteration, *Agree* is called to exchange values among the members of the local fault group. The member's values are then checked according to the consistency, feasibility, and convergence conditions. The additional stopping rule check is also employed to completely constrain Byzantine resource behavior.

Initialize:
 Start with initial labeling assignment
  $\vec{p}^{\,0}$ ε feasible labelings.
 $k \leftarrow 0$
 loop

 /* reliably send/receive neighbor's iteration values */
 $\vec{p}_j^{\,k}, \vec{s}_j^{\,k}, \vec{u}_j^{\,k} \leftarrow agree(\vec{p}_i^{\,k}, \vec{s}_i^{\,k}, \vec{u}_i^{\,k})$

/* Start of Predicate Φ */

 for each neighbor of P_i : P_j do begin
  for $\lambda = 1$ to m **do begin** /* Φ_F */
   if $(\sum_{\lambda=1}^{m} p_j^k(\lambda) \neq 1$ **or** $p_j(\lambda) \notin [0,1])$ **then**
    report error - inconsistent $p_j^k(\lambda)$;
  rof;
  if $(\sum_{\lambda} u_j(\lambda) \neq 0)$ **then** report error - infeasible $u_j^k(\lambda)$;
  if $((\vec{s}_j^{\,k} - \vec{u}_j^{\,k}) \cdot \vec{u}_j^{\,k} \neq 0)$ **then**
   report error - infeasible $\vec{u}_j^{\,k}$ *or* $\vec{s}_j^{\,k}$;
  if $(\|\vec{u}_j\| \neq 1)$ **then** /* Φ_P */
   report error - nonconvergent $\vec{u}_j^{\,k}$;
  if $(\varepsilon^k < \varepsilon_0)$ **then**
   {declare P_j as stopped, result is $\vec{p}^{\,k_j}$; **break**;}
  if $(\vec{s}_j^{\,k} - \vec{s}_j^{\,k} - 1 < \varepsilon^k)$ **then** /* Φ_P */
   report error - nonconvergent $\vec{s}_j^{\,k}$;
 rof;

/* End of Predicate Φ */

 /* nearest neighbor updating */
 $s_i^k(\lambda) = \sum_j \sum_{\lambda'} r_{ij}(\lambda, \lambda') p_j^k(\lambda')$

 /* Find a feasible gradient direction */
 $\vec{u}^{\,k} = projection\text{-}operator\ (\vec{p}^{\,k}, \vec{s}^{\,k})$

 if $(\vec{u}^{\,k} = 0)$ **break;**

 /* h is some small positive value */
 $\vec{p}^{\,k+1} \leftarrow \vec{p}^{\,k} + h\vec{u}^{\,k}$

 $k \leftarrow k+1$

pool

Figure 8.6. Reliable relaxation labeling algorithm RL_R

for each processor P_i

8.4 CHAPTER SUMMARY

Performance analysis of the reliable algorithm follows the modeling of Chapter 6. The results do not differ significantly from the performance data presented there and are not repeated here. While the mathematical theory is well developed in terms of convergence of the algorithm, it is not rich enough to support the type of analysis presented in Chapter 7. This modeling is an area for further study.

Parallel relaxation labeling for computer vision differs from simple matrix relaxation in that both the calculations are more complex and that a more rich set of natural constraints exists. Both problems, however, yield equally satisfactory constraint predicates. The next chapter details reliable solution of a completely different type of problem, that of parallel sorting.

Chapter 9

Parallel Bitonic Sorting

Sorting in the DMMP environment is not likely to be the sole application but rather a sub-problem of some other parallel applications. This distinction is important for two reasons. First, since the data is already in the node processors of the DMMP, there is no central point through which all data will pass or had passed. Since no such central point exists, there is no opportunity for centralized fault diagnosis; therefore, the fault-detection problem must be solved in a distributed manner. A second, more pragmatic, consideration is that if the data must be loaded from the external environment, it may be more efficient to simply sort the data sequentially in the host rather than incur the communication cost necessary to distribute/collect the data to/from the nodes.

The target DMMP interconnection topology considered in this chapter is the popular hypercube topology. The topology of an n-dimensional hypercube is a graph $G(P,E)$ with $N=2^n$ vertices called nodes labeled $P_0,P_1,P_2,...,P_{N-1}$. An edge $e_{i,j} \in E$ connects P_i and P_j if the binary representations of i and j differ in exactly 1 bit. If we let this bit position be k, then $P_i=P_{j \oplus 2^k}$. Thus, in an n-dimensional hypercube, each processor connects to n neighboring processors. Connections between the host and nodes are mainly used for program/data downloading and result uploading and are not represented in G.

9.1 BITONIC SORTING

Recall the sorting definition from Chapter 4. The assertion Φ_F suggested by Theorem 4-1 can be used in the sequential environment to verify the output of sorting procedure *S*. However, in the parallel environment, in general, neither the input list **I** nor the output list **O** is available to a processor implementing the assertion. Furthermore, the assertion (in general) is only applicable at the termination of the sorting procedure. There is no opportunity to flag an error that occurs earlier then the termination phase in *S*. These difficulties with general sorting foster consideration of a particular parallel sorting algorithm, the *Bitonic Sort* [Batc68].

The bitonic sort algorithm was introduced as a parallel sorting algorithm that can take advantage of interconnection topologies such as the perfect shuffle and hypercube. For our purposes the algorithm has properties that make it amenable to assertion development in the parallel environment. Furthermore, there exists a bitonic sort algorithm that maps directly to a hypercube topology.

The general idea of a bitonic sort is to build up longer bitonic sequences which eventually lead to a sorted sequence.

Definition 9-1: A Bitonic Sequence is a sequence of elements $O_0, O_1, \cdots, O_{N-1}$ such that

1) There exists a subscript i, $0 \le i \le N-1$ such that $O_0 \le O_1 \le \cdots \le O_i$ and $O_{i+1} \ge O_{i+2} \ge \cdots \ge O_{N-1}$

or

2) There exists a subscript i, $0 \le i \le N-1$ such that $O_0 \ge O_1 \ge \cdots \ge O_i$ and $O_{i+1} \le O_{i+2} \le \cdots \le O_{N-1}$

The fundamental operation in a bitonic sort is the compare-exchange, either min(x,y) or max(x,y).

Lemma 9-1: [Batc68] Given a bitonic sequence $I_0 \le I_1 \le \cdots \le I_{N/2-1}$ and $I_{N/2} \ge I_{N/2+1} \ge \cdots \ge I_{N-1}$, each of the subsequences formed by the compare-exchange

steps:

$$\min(I_0, I_{N/2}), \min(I_1, I_{N/2+1}), \cdots \min(I_{N/2-1}, I_{N-1})$$

$$= O_0, O_1, \cdots, O_{N/2-1}$$

and

$$\max(I_0, I_{N/2}), \max(I_1, I_{N/2+1}), \cdots \max(I_{N/2-1}, I_{N-1})$$

$$= O_{N/2}, O_{N/2+1}, \cdots, O_{N-1}$$

is bitonic with the property that $O_i \leq O_j$ for all $i = 0, 1, \cdots, N/2-1$ and $j = N/2, N/2+1, \cdots, N-1$.

Note that the midpoint of the sequence need not be $N/2$. However, assume that the midpoint is $N/2$ and $N = 2^k$ for some k. Pictorially, this splitting and merging is shown in Figure 9.1. It is easy to see that a bitonic sequence of length 2^k can be sorted by recursively applying the compare-exchange operation k times.

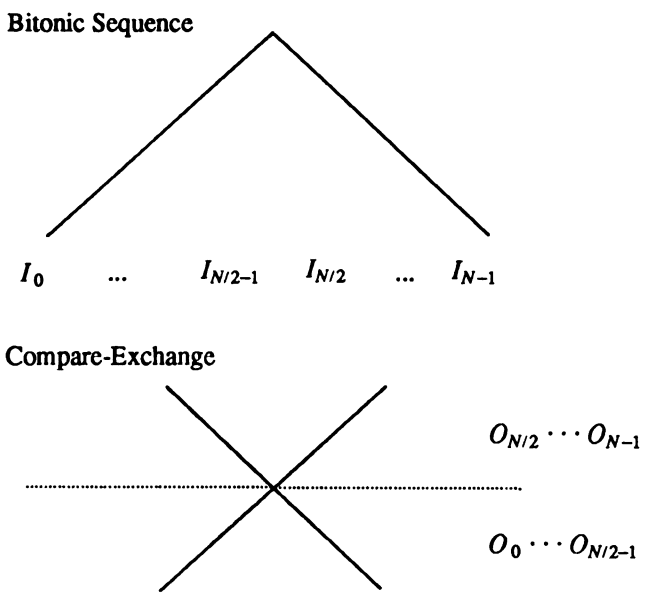

Figure 9.1. Compare-Exchange Step

Since each compare-exchange involves only a comparison between elements whose subscripts differ on only one bit and the number of elements is always 2^k, if we have one element per processor, then the bitonic sort can be easily implemented on a hypercube of dimension $n=\log_2 N$ [Quin87]. The algorithm is shown in Figure 9.2 as Algorithm S_{NR}.

```
/*
 * S_NR, executed by node node, 0≤node≤N-1
 * Local starting value in a
 */

Procedure S_NR

  for i:=0 to n-1 do
    {for j:=i downto 0 do
      {d:=2^j;
      if (node mod (2d)<d)
        {read into data from node+d;
        if (node mod 2^(i+2)<2^(i+1))
          {b := max(data,a);a := min(data,a);}
        else
          {b := min(data,a);a := max(data,a);}
        write from b to node+d;}
      else /* Send to neighbor - we are inactive this iteration */
        {write from a to node-d;read into a from node-d;}
      }   /* End for j */
    }   /* End for i */
```

Figure 9.2. Algorithm S_{NR}

The completion of each iteration of the **for** *i* is called a *stage*.

Lemma 9-2: Algorithm S_{NR} produces a bitonic subsequence in each subcube of size 2^{i+2} at the end of stage i given bitonic subsequences of length 2^{i+1} at the start of step i if no errors occur.

Proof: Proof is by induction on i, the subcube index under consideration. Initially we are given degenerate bitonic sequences of length 2. For $i=0$, since $d=2^0=1$, by Lemma 9-1 a single parallel compare-exchange step forms two bitonic sequences of length 1 in each subcube of dimension 2, either ascending or descending based on the subcube indices. This forms bitonic sequences of length $2^{i+2}=4$ in each 2-cube.

Assume that for $i=k$, algorithm S_{NR} has produced a bitonic sequence in each subcube of size 2^{k+2} at the end of step k. Then for $i=k+1$, since $d=2^{k+1}$, a single compare-exchange step, by Lemma 9-1, forms two bitonic sequences each of length 2^{k+1}, one high and one low. Since the next iteration of j is a bitonic sort on these two subcubes of dimension 2^{k+1}, by the inductive hypothesis at the termination of step $k+1$, the subcube of size 2^{k+3} contains a bitonic sequence. □

Theorem 9-1: Algorithm S_{NR} sorts a list of items arranged in a cube of size $N=2^n$.

Proof: Immediate from Lemma 9-2 by iteration of the **for** i loop from 0 up to $n-1$ in algorithm S_{NR}. □

Since the code executed by the inner **for** j loop is clearly $O(1)$, the runtime of this parallel algorithm is $O(n^2)=O(log_2^2 N)$.

Our goal in development of this algorithm is to implement a system which can sort in parallel and never produce an incorrect output. To perform this task, the notion of a fault must be clearly defined.

9.2 FAULTY BEHAVIOR

Definition 9-2: A *fault* is a deviation from the correct sequence of a calculation that produces an incorrect result. A system is said to be *fail-stop* if, upon the occurrence of a fault, no output is produced and the system simply halts. A component is said to exhibit *Byzantine* [LaSP82] behavior if, upon the occurrence of a fault, the effect of the fault on the overall calculation is to produce an incorrect result. Furthermore, this will be done in the most malicious manner possible. A node P_i is said to be *faulty* under the following cases:

1) A fault occurs in processor P_i and any number of its incident communication links.
2) A fault occurs in one or more of P_i's incident communication links but not in P_i. There are two cases.

a) The fault occurs in exactly one incident communication link $e_{i,j}$. If a fault has also occurred in P_j, then P_j is faulty. If no fault has occurred in P_j, then arbitarily P_i is declared faulty and P_j is declared non-faulty.

b) The fault occurs in two or more incident communication links. Then P_i is declared to be faulty.

The following assumptions are made in the development of the parallel sorting algorithms.

Environmental Assumptions:

1) Inter-node communications and processors are subject to Byzantine faults.
2) The host processor is reliable as are the links from the host processor to each node.
3) Message transmission is over point-to-point links and no atomic broadcast exists.
4) The absence of a message can be detected and constitutes an error.
5) All nodes are non-faulty at initiation of the algorithm and remain non-faulty through the first message exchange.

Definition 9-3: The home subcube $SC_{i,j}$ of dimension i of a processor P_j is the subcube of size 2^i that begins with processor P_k, $k=j-(j \bmod 2^i)$ and includes all processors through P_l, $l=j-(j \bmod 2^i)+2^i-1$. Let $SC^S_{i,j}$ denote the index k and $SC^E_{i,j}$ denote the index l. If the bitonic sequence in $SC_{i,j}$ is comprised of an ascending sequence followed by a descending sequence each of length 2^{i-1}, then the flag *ascending* is true and false otherwise.

9.3 RELIABLE BITONIC SORTING ALGORITHM DEVELOPMENT

The reliable bitonic sorting algorithm S_{FT} is given in Figure 9.4. The individual components of the constraint predicate Φ_P, Φ_F, Φ_C corresponding to the progress, feasibility, and consistency predicates subclasses are discussed in the following paragraphs.

```
/* S_FT, executed by node, 0≤node≤N−1 - Local starting value in a */
Procedure S_FT
  LBS[j]:=a; lmask:=2^j;
  for i:=0 to n−1 do
    {limit:=min(SC^E_{i+1,node},N);
     for j:=i downto 0 do
       {d:=2^j;
        if (node mod (2d)<d)
          {read into (data,lbuf) from node+d;
           lmask:=Φ_C(LBS_i,j,lmask);
           if (node mod 2^{i+2}<2^{i+1})
             b := max(data.a,a);a := min(data.a,a);
           else
             b := min(data.a,a);a := max(data.a,a);
           data:=(a,b);
           write from data,LBS to node+d;}
        else
          {data.a:=a;
           write from data,LBS to node−d;
           read into (data,lbuf) from node−d;
           lmask:=Φ_C(LBS_i,j,lmask);
           (a,b):=data;}
       }   /* End for j */
  /* Verify both the validity of the sorted sequence in LBS with respect
     to LLBS and the bitonic nature of the sequence in LBS */
    if (i≠0)
      if (not bit_compare(LLBS_i,LBS_i)
        signal ERROR to host;
      for m:=SC^S_{i,node} to limit /* Update LLBS */
        LLBS[m]:=LBS[m];
    LBS[node]:=a;lmask:=2^node;
    }   /* End for i */
  /* Verify Last Stage by pure exchange of final LBS */
   i:=n-1;
   for j:=i downto 0 do
     {d:=2^j;
     if (node mod (2d)<d)
       read into lbuf from node+d;lmask:=Φ_C(LBS_i,j,lmask);write from LBS to node+d;
     else
       write from LBS to node-d;read into lbuf from node-d;lmask:=Φ_C(LBS_i,j,lmask);
     }   /* End for j */
     if (not bit_compare(LLBS_i,LBS_i)
        signal ERROR to host;
}
```

Figure 9.4. Algorithm S_{FT}

9.3.1 Progress

The progress component Φ_P ensures that algorithm termination will occur and that at each testable step of the solution, the state of the solution advances to the goal or final solution of the problem. If this progress is not made, then faulty behavior may indefinitely postpone the solution - any solution - even an incorrect solution.

For iterative convergent problems, the progress component involves reduction of error. For the bitonic sorting problem, the number of steps is known a priori to all participants in the algorithm. Thus any early termination is considered an error. The testable step for Φ_P is at the bottom of the outer loop i in algorithm S_{NR}. By Lemma 9-2 this must be a bitonic sequence of length 2^{i+1}.

The progress test Φ_P is given in Figure 9.5a. The sequence BS_i is a bitonic sequence of length 2^{i+1} in $SC_{i+1,node}$.

Procedure $\Phi_P(BS_i)$

```
for k := SC^S_{i,j} to SC^E_{i,j}
  if (ascending)
    if (BS[k+1]<BS[k]) then ERROR;
  else
    if (BS[k+1]>BS[k]) then ERROR;
if (i ≠ n)
  for k := SC^S_{i,j}+2^i to SC^E_{i+1,j}
    if (ascending)
      if (BS[k+1]>BS[k]) then ERROR;
    else
      if (BS[k+1]<BS[k]) then ERROR;
```

Figure 9.5a. Φ_P - Progress Component

9.3.2 Feasibility

Each testable result must remain within the defined solution space H obtained from the natural constraints of the problem. The natural constraint here, and indeed the reason for the choice of the bitonic sort procedure, is that at each stage i of the computation, the bitonic sequence formed must contain only the elements to be sorted, no more, no less.

Definition 9-4: The bitonic sequence $LLBS_i$ is a bitonic sequence of length 2^i resulting from stage $i-2$ of algorithm S_{FT}. The bitonic sequence LBS_i is a bitonic sequence of length 2^{i+1} resulting from stage $i-1$ of algorithm S_{FT}.

The feasibility test Φ_F is given in Figure 9.5b. Note the "C" conditional statement syntax in the **for** loop. In this notation, x?y:z indicates that if x is true, execute y, otherwise execute z.

Procedure $\Phi_F(LLBS_i, LBS_i)$

$l := SC^S_{i,node}$;
$u := SC^E_{i,node}$;
for m := ascending?$SC^S_{i,node}$:$SC^E_{i,node}$;ascending?**to**:**downto** ascending?$SC^E_{i,node}$:$SC^S_{i,node}$

  if (LBS_i[m]=$LLBS_i$[l] && $l \le 2^{i-1}+SC^S_{i,j}$)
    $l := l+1$;
  else if (LBS_i[m]=$LLBS_i$[u] && $m \ge 2^{i-1}+SC^S_{i,j}$)
    $u := u-1$;
  else
    ERROR;

Figure 9.5b. Φ_F - Feasibility Component

9.3.3 Consistency

In the case of bitonic sorting, the bitonic subsequence must be distributed to checking processors. Since it is entirely possible for a Byzantine faulty processor to send different versions of the same message to different checking processors, each version of which satisfies the feasibility test locally but is incorrect globally. This situation is said to be *inconsistent*. To achieve consistency, we require that each processor "hears" the same version of a message, in this case a bitonic subsequence. This can be accomplished by sending two copies of each bitonic sequence via vertex disjoint paths to each checking processor. The message routing is constructed in such a way that each message must

pass though processors that are capable of checking parts of each message locally. The intersection of these tests, given the same input message, forms a global test. When the locally checked messages meet at a checking processor, they must contain the same information. If not, then the message has been altered to satisfy each individual processor locally. This situation is then detected at the checking processor, and an error is flagged.

The consistency test Φ_C is given in Figure 9.5c. Note that in keeping with the application oriented paradigm, the test for faulty behavior is closely intertwined with the actual message delivery of the last bitonic sequence LBS_i. *lmask* is initially set to the binary representation of j, and *source* is the address of the sender of the message in *lbuf*.

Briefly the concept of the fault-tolerant bitonic sorting algorithm is that as we build up longer bitonic sequences, we (1) check that these sequences are bitonic and (2) check that these bitonic sequences are permutations of the the earlier, shorter, bitonic sequences. The communication of a bitonic sequence from stage i is "piggybacked" in the communication that occurs naturally at stage $i+1$. This affords the resulting fault-tolerant algorithm no increase in communication complexity over the non fault-tolerant algorithm S_{NR}.

Definition 9-5: A bitonic (sub)-sequence LBS_i is *complete* if for each $I_j \in LLBS_i$ there exists a permutation Π on the elements of $LLBS_i$ to the lower or upper half of LBS_i determined by the range of $SC_{i-2,j}$ that is bijective.

Lemma 9-3: Algorithm vect_mask(i,j,k) provides a bit_vector in which a 1 in bit position l indicates that $LBS_i[l]$ has been collected from node l in a message exchange at node k from iteration i and one or more of the iterations $j, j-1, \cdots, 0$.

Proof: By induction on j, we show that the bit vector returned by vect_mask represents the actual message traffic of algorithm S_{FT} (and S_{NR}). If $i=j$, then this is the start of stage i of the message exchange, and mask is set to processor P_k and $P_{k \oplus 2^j}$.

Procedure $\Phi_C(LBS_i, j, lmask)$

```
/* y>>x is a bitwise right shift of y by x bits */

  limit=min(SC^S_{i+1,j},N)
  mask:=vect_mask(i,j,source);
  omask:=mask;
  mask:=mask>>SC^S_{i+1,j};
  lmask:=lmask>>SC^S_{i+1,j};

  for k:=SC^S_{i+1,j} to limit
    {if (mask&01 && !(lmask&01))
     LBS[k]:=lbuf[k];
     else if (mask&01 && lmask&01)
     if (LBS[k] ≠ lbuf[K])
       ERROR;
     mask:=mask>>1;
     lmask:=lmask>>1;}
   return(omask);

Procedure vect_mask(i,j,node)
  d=2^j;
  if (j=i)
    if (node mod (d<<1)<d)
      mask=1<<node | 1<<node+d;
    else
      mask=1<<node | 1<<node-d;}
  else
    if (node mod (d<<1)<d)
      mask=vect_mask(i,j+1,node+d) | vect_mask(i,j+1,node);
    else
      mask=vect_mask(i,j+1,node-d) | vect_mask(i,j+1,node);
  return(mask);
}
```

Figure 9.5c. Φ_C - Consistency Component

Assume that vect_mask($i,l+1,k$) returns a correct (as in the lemma statement) mask. Then for $j=l$, the mask returned is the $j+1$ mask for P_k or'ed with the $j+1$ mask for $P_{k \oplus 2^j}$, each of which is the correct mask. □

Lemma 9-4: Algorithm bit_compare detects non-bitonic LBS_i and non-complete LBS_i with respect to $LLBS_i$ given a bitonic $LLBS_i$ as input.

Proof: Trivial from the previous discussion and bit_compare code. □

Procedure bit_compare($LLBS_i$,LBS_i)

$\Phi_P(LBS_i)$;
$\Phi_F(LLBS_i,LBS_i)$

Lemma 9-5: At the end of stage i, at least one processor in $SC_{i-1,j}$ can detect an error made by processor P_j that results in either (1) a non-bitonic LBS_i or (2) if the the maximum number of faulty nodes in $SC_{i-1,j}$ is 1, a non-complete LBS_i given a bitonic, complete $LLBS_i$.

Proof: The proof is by induction on i, the step count. For i=0, each P_j,P_{j+1} for j even contains the actual correct initial values I_j,I_{j+1} in LBS_0. This forms a complete bitonic sequence of length 2. This then becomes $LLBS_0$ at the bottom of step i.

Assume that LBS_k has been verified complete and correct with respect to $LLBS_k$ at the end of step k. Then during step k+1, LBS_{k+1} is filled by partial LBS_{k+1}'s according to the bit sequence of vect_mask(k+1,j). Since each element considered for inclusion in LBS_{j+1} is reported to at least one processor through two vertex disjoint paths in graph G, the effects of a single faulty relay are limited to one of these paths. If the two candidate elements differ, an error is signaled. Thus if the sender is faulty, it must send identical values along both paths. If these values destroy the bitonic nature of LBS_i or if they are not complete with respect to $LLBS_i$, then, by Lemma 9-4, bit_compare will flag an error. Otherwise, no error has occurred. □

An example of S_{FT} is shown for n=3 in Figure 9.6. The list to be sorted, {10,8,3,9,4,2,7,5} is stored in processors P_0–P_7. Note that for stage 1, the LBS and LLBS are shown only for $SC_{1,2}$ and $SC_{1,6}$ and for stage 2, only $SC_{2,0}$ is shown.

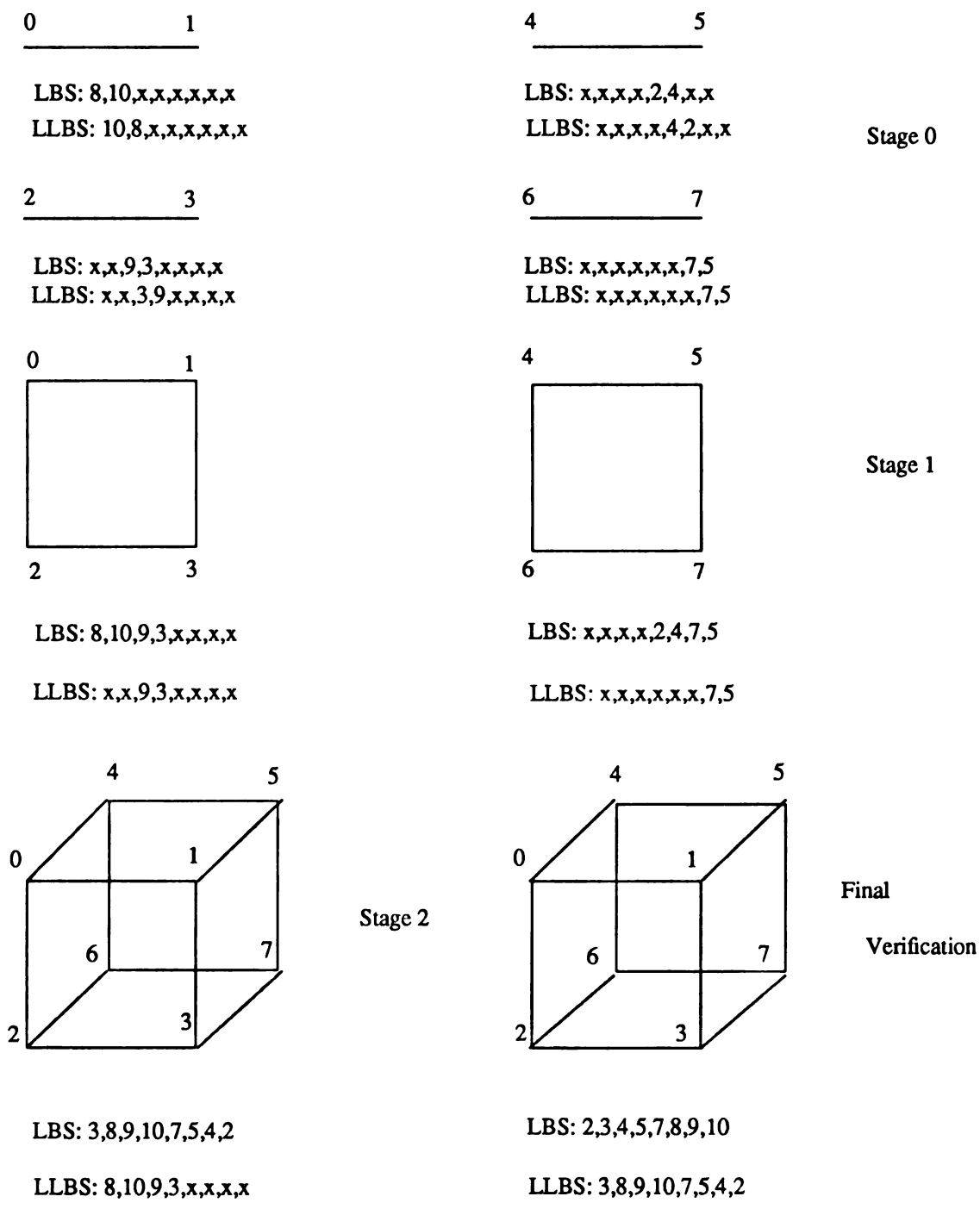

Figure 9.6. Example of S_{FT} for n=3.

9.4 ERROR COVERAGE AND RESILIENCE

Analysis of the expected error coverage is key for any fault-tolerant algorithm.

Theorem 9-2: Algorithm S_{FT} produces either a correct bitonic sort or stops with an error in the presence of one faulty node.

Proof: By application of Lemma 9-5, at each step i of the for loop in S_{FT}, each bitonic sequence is verified. The final extra stage verifies that last sequence. Since we are allowed one faulty node per $SC_{i,j}$ (i now = n in bit_compare), a processor P_j can detect any faulty behavior. □

As Theorem 9-2 shows, the constraint predicate Φ formed by Φ_P, Φ_F, Φ_C detects all errors from the Byzantine fault class committed by one faulty processor. Thus the reliable bitonic sort algorithm is fail-stop using components which may fail in Byzantine ways. The result of the calculation is either completely correct, or the entire system halts with an error condition. We are guaranteed that under a single processor failure, we will never receive an incorrect sorting result. Furthermore, unless an error occurs, the host is never involved in the calculation. This is ideal from a performance standpoint since, as mentioned earlier, the host can become a bottleneck.

Clearly, there will be a performance penalty to pay for the increased reliability of algorithm S_{FT} over the unreliable S_{NR}. One may question how much overhead is introduced and whether it might be better to simply send all the data to the host, let the host sort the data, and return the final result to the node processors. Another possibility is to send all the data to the host, sort the data in the node processors, and send the results to the host for verification. As we show in the next section, the performance of both these possibilities becomes unreasonable for even moderately large problems.

9.5 TIME AND SPACE COMPLEXITY

As mentioned in Section 2, algorithm S_{NR} has a time complexity of $O(log_2^2 N)$. Sequential sorting, on the other hand, has a lower bound of $O(N\log_2 N)$. The

communication complexity (since the data must be transferred from the nodes to the host) is $O(N)$ for sequential sorting. The communication complexity for S_{NR} is $O(log_2^2 N)$. Thus it is expected that the fault tolerant algorithm S_{FT} will have time/communication complexity between these two.

Lemma 9-6: Algorithm vect_mask(i,j) has time complexity of $O(2^{i-j})$.

Proof: Let the running time of vect_mask(i,j) be $T_{VM}(i,j)$. If $i=j$, then four logical bit operations are performed and $T_{VM}(i,i)=O(2^0)=O(1)$. If $i>j$, then two executions of vect_mask($i,j+1$) are performed. Thus we have the linear recurrence

$$T_{VM}(i,j) = 2T_{VM}(i,j+1)$$

Solving this yields:

$$T_{VM}(i,j) = 2^{i-j}T_{VM}(i,i)$$

$$= O(2^{i-j})$$

□

Lemma 9-7: Algorithm bit_compare() has a time complexity of $O(2^i)$ for a calling node k at step i.

Proof: Finding the size and location of the subcube is $O(1)$ by simple application of Definition 9-3. Verifying the sorted nature of each half of LBS_i can be done in $O(2^i)$ time. Verification of the completeness of LBS_i with respect to $LLBS_i$ can be done in $O(2^i)$. Thus the total time complexity is $O(3\cdot 2^i)=O(2^i)$.

Lemma 9-8: Algorithm $\Phi_C(i,j)$ has a time complexity of $O(2^{j+1}+2^{i-j})$.

Proof: There are at most 2^{j+1} non-zero entries reported by vect_mask at step j plus the time for vect_mask(i,j) of $O(2^{i-j})$ by Lemma 9-6 gives the bound. □.

Theorem 9-3: Algorithm S_{FT} has a computational time complexity of $O(N)$ and a communication complexity of $O(log_2^2 N)$.

Proof: For a particular value of i, j ranges from 0 to i. By Lemma 9-8, each Φ_C(i,j) has complexity $O(2^{j+1}+2^{i-j})$. Since each iteration contains time for 2 Φ_C iterations

(because the computations cannot be overlapped), we have

$$2\sum_{j=0}^{i} 2^{j+1}+2^{i-j}=2\cdot 2^{i+2}+2^{i+1}=O(2^{i+3})$$

By Lemma 9-7, bit_compare has complexity 2^i and updating LLBS takes 2^i; so for a single iteration of i, we have a run time of $O(2^{i+3}+2^i+2^i)$. Summing over all i from 0 to $n-1$ and adding in the final verification step, we have

$$\sum_{i=0}^{n-1}(2^{i+3}+2^i+2^i)+2^{n+2}=2^{n+3}+2\cdot 2^{n-1}+2^{n+2}=O(2^{n+3})$$

Since $n=\log_2 N$, we have the run time of $S_{FT}=O(2^{\log_2 N+2})=O(N)$

The communication complexity of the main loop, as in S_{NR}, is $O(log_2^2 N)$. The final verification stage adds $\log_2 N$ communication so the order of S_{FT} remains unchanged from S_{NR}. □

For comparison purposes, a sequential "sorting" algorithm was constructed for the host. Sort is quoted since we implement this "sort" as a single **if** statement executed $N\log_2 N$ times to achieve the theoretical minimum. $O(N)$ communication is required to send/receive the sorted data. Additionally, a sequential verification was constructed. In this algorithm, the initial data is sent to the host, sorted by the node processors, and the sorted data also sent to the host. The host then implements Theorem 4-1 to verify the results. This takes $O(N)$ communication complexity. It also takes $O(N\log_2 N)$ computational complexity since the matching of the ordered and unordered list becomes equivalent to finding a permutation in the sense of Definition 4-1. Thus, for the following discussion, the best sequential algorithm is $O(N\log_2 N)$.

The algorithms S_{NR}, S_{FT}, and a sequential sort were implemented on a 32 nodes of a 64 node Ncube DMMP [HMSC86] to sort 32-bit integers into ascending order (implementation constraints forced consideration of the smaller subcube). Timings were obtained for for all three algorithms for problem sizes of 4, 8, 16, and 32 nodes. This is shown in Figure 9.7.

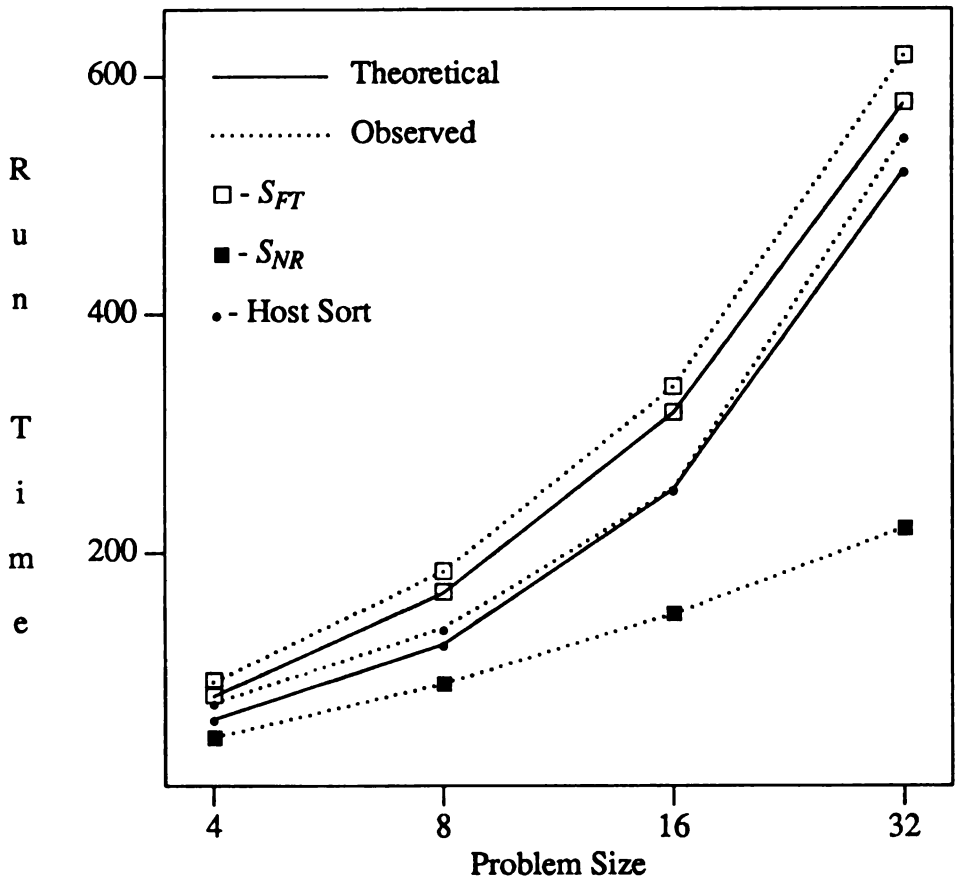

Figure 9.7. Sorting Time Comparisons

The execution (observed) results are inconclusive since the cube we have available is very small. Asymptotically, we certainly expect S_{FT} to be more efficient than the host sort, but the constant multiplier in the run time order dominates for these small problem sizes. Measurement of the running time for each component of the two algorithms yields the following table (measured in clock ticks).

Algorithm	Communication Time	Computation Time
S_{FT}	$8\log_2^2 N + .250N$	$.72(16N)$
Sequential	$14N$	$0.45N\log_2 N$

This behavior is plotted as (Theoretical) in Figure 9.7. Comparison with the observed values indicates this approximation is close to the actual run time for smaller cube sizes. In the projected run times of Figure 9.8, S_{FT} rapidly becomes more efficient for the size of cubes that we are concerned with in a real DMMP application. The portion of this plot covered by Figure 9.7 is highlighted in the lower left corner of Figure 9.8. These projected run times indicate that the penalty paid for fault tolerance in parallel sorting is less than the cost for sequential sorting in the host.

9.6 CHAPTER SUMMARY

This chapter has presented a fault-tolerant parallel sorting algorithm developed using the application oriented fault-tolerance paradigm. The algorithm is tolerant of one faulty processor node, or faults in a single processor's incident communication links.

The addition of reliability to the sorting algorithm results in a performance penalty. While the experiments on the small cube available were unable to demonstrate that fault tolerant sorting is more efficient than simply sorting in the host, asymptotically the developed fault tolerant algorithm is less costly than host sorting. Experiments on a larger cube are necessary to verify this behavior. Additionally, performance of the three algorithms under bitonic sort/merge is interesting to study. In this algorithm, each node has not one element, but a small, however, non-negligible number. The sequences are sorted locally and then merged in parallel to form bitonic sequences. This bitonic sort/merge procedure is an easy extension of S_{NR}. To extend S_{FT} to this sort/merge case

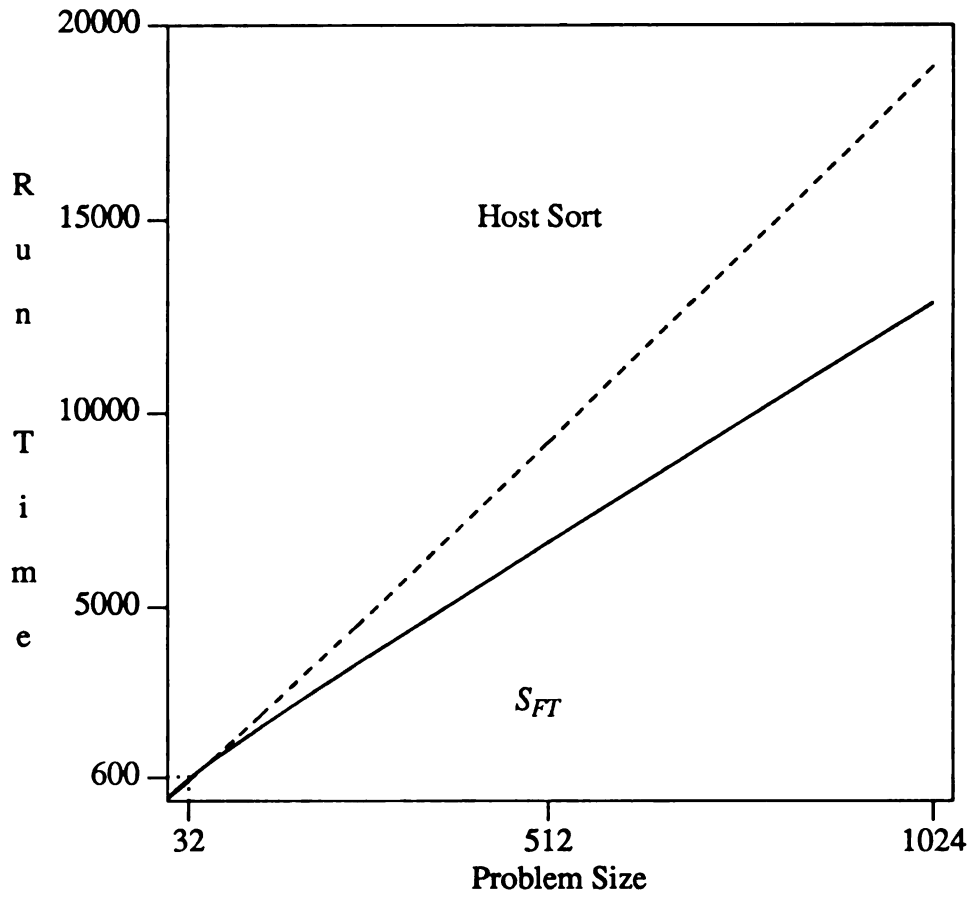

Figure 9.8. Projected Sorting Time Comparisons - Large Systems

is more difficult. Simply growing the present algorithm will undoubtedly result in unacceptable run time and space requirements. To reduce the run time complexity, the computation/communication ratio will shift towards more communication of shorter messages.

This chapter has demonstrated that the application oriented fault tolerance paradigm is applicable to problems of a non-iterative nature. Again, all that is necessary for successful algorithm development is a sufficient set of natural problem constraints; in this case the bitonic nature of the intermediate results.

Chapter 10

Summary and Directions for Future Research

This chapter summarizes the major contribution of this thesis and outlines directions for future work.

10.1 SUMMARY OF MAJOR CONTRIBUTIONS

This work has been motivated by a need for both hardware and software reliability in large scale DMMP systems. From a hardware perspective, the thousands of components in the DMMP will result in low overall system reliability [McNi88a]. Thus, system-level techniques must be employed to achieve reliable system operation. However, traditional system level techniques such as N-Modular Redundancy are too expensive to implement. The problem of providing reliable software is not as easy to quantify but is easy to see its need. The detection (and recovery from) design/coding faults is at the heart of software fault-tolerance. However, to apply software fault-tolerance techniques to a DMMP requires a different approach [McNi88b]. Both of these problems may be solved by the unified method proposed by this research, the application oriented fault-tolerance paradigm.

The application oriented fault-tolerance paradigm is a completely novel way to provide system level reliability. It is necessarily an application based software approach to fault-tolerance. Thus, most of the "system-oriented" issues must be isolated from the applications programmer. These system oriented routines include reliable broadcast and detection support and form the lower layers of the reliable parallel processing model.

The application and its interface to reliability form the higher layers of this model. The level of coupling between these two layers is low. The applications programmer then must specify only an abstraction of fault-tolerance that is based on the application at hand.

A systematic way of generating executable assertions for the DMMP environment is essential to the application oriented fault-tolerance paradigm. The three predicate subclasses of progress, feasibility, and consistency provide a systematic technique to formulate the constraint predicate. The constraint predicate is embedded in the actual program for both hardware and software fault-detection. Thus, an application with an embedded constraint predicate constrains processor and software behavior to remain within predefined limits. This is an effective paradigm for many types of parallel computing applications as demonstrated by its application to the "bread and butter" algorithms of parallel computation, namely, matrix iterative analysis [McNi87a, McNi88b], computer vision [McNi88a], and parallel sorting [McNi88c]. The only known restriction on applicability of the constraint predicate paradigm is that the applications problem contain sufficient natural constraints.

Since the components of the constraint predicate are executed by a number of reliable processors, the diagnosis is guaranteed to locate the faulty component. The constraint predicate relys upon the distributed diagnostic basis to communicate its test results. There are clearly two different possible approaches to a distributed diagnostic basis. The tight coupling with the application oriented fault-tolerance paradigm favors the use of a masking basis such as Byzantine Agreement and rejects the use of syndrome testing. The algorithm Vector Byzantine Agreement [McNi87b] functions as an efficient diagnostic basis for the DMMP environment under Byzantine fault conditions. Syndrome testing can never be complete in this manner, only correct [McEN88].

There will be a run time penalty to be paid for the use of reliability. There is a tradeoff based on expected run time between using reliability or running an unreliable

algorithm. For very short problems, it is better to run with no reliability than to incur the overhead of a reliable parallel algorithm. For larger problems there is a crossover point at which it is more effective to run with the reliability penalty and be guaranteed a correct solution within a bounded run time. Both asymptotically as well as practically, since the reliable solutions are only a linear factor of the fault free run time as opposed to an exponentially growing run time without reliability, the reliability methods developed in this research provide the necessary cost effective solution to the problem of reliability in parallel computation.

Not only is run time an important metric in judging the effectiveness of any reliability scheme but also the expected coverage of errors. As shown in the error coverage section of matrix iterative analysis, seventy-five percent of all errors can be detected immediately, and total coverage can be achieved at algorithm termination. Similar results exist for bitonic sorting, the reliable algorithm never allows an incorrect result to be produced.

In summary, this work has accomplished its objectives. It is clear that this area will continue to receive much study. The following sections outline the direction this work should take.

10.2 Configuration Control

This thesis has presented a new paradigm to provide both hardware and software reliability in a faulty large scale DMMP and has concentrated mainly on the fault-detection aspect of the problem. However, to provide a reliable system, the problem of continuing execution in the presence of failed components must be addressed as well as the problems of providing reliable Remote Procedure Calls (RPC) in the unreliable environment.

The RPC scheme proposed by [BiNe84] has a drawback in the unreliable environment. From the application level, RPCs are treated as simple function calls. If the called

process fails, the procedure call simply blocks forever. To alleviate this problem, timeouts are employed. Necessary bounds for timeouts in the DMMP environment are (1) A bound on the maximum processor clock drift Ψ and (2) A bound on the maximum communication delay Δ. The problem with specification of these values is that they are configuration dependent. The application programmer level should not be concerned with even the existence of these values. Moreover, these values are primarily used by the distributed diagnostic basis. Thus, in the reliable parallel processing model, the *Configuration Control* section of Figure 1.4 (p. 17) is responsible for computing these values.

Dynamic redundancy refers to the ability of a system to continue to function in the presence of failed components. Masking of errors is limited as a fault-tolerance technique. Consider again IBM's GF11 computer [Ager88] whose operational design parameters dictate single problem run time on the order of a year in length. If only fault-masking is used as a fault-tolerance technique, eventually enough elements will fail to invalidate fault masking. Furthermore, masking of errors also has the difficulty that the reporting and knowledge of fault occurrences is also masked [Hopk77].

Thus, instead of masking faults, we wish to remove the faulty component from the system and reconfigure around it. Additionally, as a maintenance activity, the faulty component should be repaired or replaced. In [YaHa84], this recovery consists of three phases; fault diagnosis, system reconfiguration, and operational recovery.

10.2.1 Reconfiguration

Centralized reconfiguration control is the easiest to implement. However, this technique is unacceptable in a distributed system because it introduces a single point of failure. We instead turn to decentralized reconfiguration control.

There are two approaches to reconfiguration. The fault group may attempt to replenish its ranks with a spare. This is advantageous as the spare can pick up the duties

of the failed processor and the algorithm execution can continue with no performance degradation. The drawback to this approach is that standby spares must be made available and that some hardware mechanism must exist as in the MPP [Batc80] to map the spare component into the system topology. The alternative to this approach is to make use of the applications oriented environment.

In the applications environment, since we have the concept of a local fault group, it is possible that reconfiguration can be done in a local distributed manner. The reconfiguration involves not replacing the hardware component, but remapping portions of the application to fault-free components. This task, in the reliable parallel processing model, is handled by the configuration control layer.

The function of the configuration control layer is also to recompute the Δ and Ψ timeout values. Even in the latest DMMP systems, the *second generation hypercubes*, the hop delay is still non-negligible. Remapping portions of the application will necessitate propagation of the new timeout values to the remaining non-faulty processors. This is to be hidden as much as possible from the applications layer. It is not hard to see that results of this research can be applied to more general usage of parallel computers that have failed components. For example, current techniques for running with a faulty hypercube system involve not executing around the failed component but instead finding a smaller completely functional subcube. However, this problem is not easily solvable and indeed is polynomially complete [DuHa88]. The reconfiguration control proposed here not attempt to find a completely functional topology but rather work within the existing topology containing failed components.

10.2.2 Recovery

Coupled with the problem of reconfiguration is operational recovery and continued execution in the presence of a changed topology due to failed and logically removed hardware components.

Recovery involves restarting the calculation after reconfiguration has occurred. Current techniques require a reliable backing store. However, in the applications environment, the information obtained by members of the fault group of a failed processor required by the constraint predicate may also be sufficient information to restart the calculation from the point of failure.

10.3 AUTOMATED CONSTRAINT PREDICATE GENERATION

The application oriented reliability paradigm currently consists of a set of basis metrics for constraint predicate extraction. These are applied at the specification phase of the software life cycle by the programmer. Ideally, the salient reliability features to compose the constraint predicate would be extracted from the applications problem in an automated fashion. While this research has made significant progress in generation techniques for a constraint predicate, the application oriented reliability paradigm is still in its infancy. Two primary reasons exist for this. The first is that while human identification of the required constraint predicate features can be comprehended, it still takes an in-depth knowledge of the application to extract the features. The second problem is that specification of the constraint predicate and its insertion into the code is a manual task. Both of these problems, however, can be addressed by Computer Aided Software Engineering (CASE) tools.

The problem of application knowledge is not as severe as it might first seem. While expert knowledge is required of a particular application, this knowledge transfers easily between similar instances of a problem. Thus it is quite feasible to work with the idea of problem classes, i.e., the class of relaxation problems, the class of sorting problems, etc. A CASE tool can consist of an expert system with various experts than can be employed based on the class of problem under consideration. Insertion of the constraint predicate can occur as a result of the same expert system.

The major remaining problem is to generate a uniform representation of the requisite abstraction of reliability information necessary for constraint predicate generation. Given a uniform representation, the tool of Automated Reasoning can be applied to generate the constraint predicate. However, automated reasoning is still very "class-based," i.e. it works for well-defined applications. Additionally, if the representation of the reliability abstraction is similar, save for a syntactic transformation, to the constraint predicate specification, then nothing is gained through automation. Thus, in the best case, while automated reasoning can provide help in specification of the constraint predicate, it is suspected that identification of the abstraction points will remain a human activity.

10.4 APPLICATION APPLICABILITY

Parallel iterative relaxation methods form a large component of all parallel algorithms. The local nature and superior natural constraints of these algorithms yield good constraint predicates. For other algorithms, the transition is not as clear. Bitonic sorting, for instance, has no such "nice" local properties. Sequential assertions, such as the sorting assertion of Randall [Rand75] cannot be implemented in the DMMP environment since no single processor has a complete view of the data space. However, through use of the constraint predicate paradigm, a reliable parallel algorithm was created [McNi88c].

It is not clear at this time exactly which problems lend themselves to application oriented reliability other than the general guideline of those with suitable natural constraints. A better classification scheme is necessary. This will continue to be a research area in the field. This clearly is an interdisciplinary study which ideally is conducted in concert with practitioners of each application field.

REFERENCES

[AdBC82] Adrion, W., Branstad, M. and Cherniavsky, J., "Validation, Verification, and Testing of Computer Software," *Computing Surveys*, Vol. 14, No. 2., June 1982, pp. 159-192.

[Ager88] Agerwala, Tilak, Invited Talk, Michigan State University, April, 1988.

[Ames77] Ames, William, *Numerical Methods for Partial Differential Equations*, Academic Press, New York, 1977.

[AnMa67] Anderson, J. and Macri, F., "Multiple Redundancy Applications in a Computer," *Proceedings of the 13th International Symposium on Fault-Tolerant Computing*, Washington D.C., January 1967, pp. 553-562.

[Aviz71] Avizienis, A. et. al. "The STAR Computer: An Investigation Into the Theory and Practice of Fault-Tolerant Computing," *IEEE Transactions on Computers*, Vol. C-20, November 1971, pp. 1312-1321.

[AyOz87] Aykanat, C., Ozguner, F., "A Concurrent Error Detecting Conjugate Gradient Algorithm on a Hypercube Multiprocessor," *17th Symposium on Fault Tolerant Computing*, Pittsburg PA, July 1987, pp. 204-209.

[Batc68] Batcher, K., "Sorting Networks and Their Applications," *Proc. of the 1968 Spring Joint Computer Conference*, vol. 32, AFIPS Press, Reston, VA, pp. 307-314.

[BiNe84] Birrell, A. and Nelson, B., "Implementing Remote Procedure Calls," *ACM Transactions on Computing Systems*, Vol. 2, No. 1, February 1985, pp. 63-75.

[BoVe82] Botta, E. and Veldman, A., "On Local Relaxation Methods and Their Application to Convection-Diffusion Equations," *Journal of Computation Physics*, Vol. 48, 1982, pp 127-149.

[DaMa84] A.T. Dahbura and G. M. Masson, "AN $O(n^{2.5})$ fault identification algorithm for diagnosable systems," *IEEE Trans. Comput.*, vol. C-33, July 84, pp. 486-492.

[DLPS86] Dolev, D., Lynch, N., Pinter, S., Stark, E., Weihl, W., "Reaching Approximate Agreement in the Presence of Faults," *Journal of the ACM*, Vol. 33, No. 3, July 1986, pp. 499-516.

[Dole82] Dolev, D., "The Byzantine Generals Strike Again," *Journal of Algorithms*, Vol. 3, 1982, pp. 14-30.

[DoSt82] Dolev, D., Strong, H., "Polynomial algorithms for multiple processor agreement," IBM Research, San Jose, 1982

[DuHa88] Dutt, S. and Hayes, J., "On Allocating Subcubes in a Hypercube Multiprocessor," *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, Pasedena, CA, January, 1988 to appear.

[DwLS88] Dwork, C., Lynch, N., Stockmeyer, L., "Consensus in the Presence of Partial Synchrony," *Journal of the ACM*, Vol. 35, No. 2, , April 1988, pp. 288-323.

[FiLP85] Fischer, M, Lynch, N. and Patterson, M., "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the Association for Computing Machinery*, Vol. 32, No. 2, April 1985, pp. 374-382.

[Garc82] Garcia-Molina, H., "Elections in a Distributed Computing System," *IEEE Transaction on Computers*, Vol. C-31, No. 1, Jan. 1982, pp. 48-59.

[Gend87] Gendreau, T. *A Unified Environment for Distributed Computing*, Ph.D. thesis, Department of Computer Science, Michigan State University, January 1987.

[GrRe86] Grunwald, D.C. and Reed, D. A., "Benchmarking hypercube hardware and software," *Technical Report*, UIUCDCS-R-86-1303, Department of Computer Science, University of Illinois at Urbana-Champaign, 1986.

[GuRa86] Gupta, R. and Ramakrishnan, I., "System Level Diagnosis in Malicious Environments," *Technical Report*, SUNY Stony Brook, Dept. of Computer Science, Oct, 1986.

[HaAm74] Hakimi, S. and Amin, A., "Characterization of connection assignment of diagnosable systems," *IEEE Trans. Comput.*, vol. C-23, Jan 1974, pp. 86-88.

[Hill85] Hillis, D., *The Connection Machine*, MIT Press, Cambridge, Mass., 1985.

[HiSt86] Hillis, D. and Steele, G., "Data Parallel Algorithms," *CACM*, Vol. 29, No. 12, , December, 1986, pp. 1170-1183.

[HMSC86] Hayes, J., Mudge, T., Stout, Q., Colley, S. and Palmer, J., "Architecture of a Hypercube Supercomputer," *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986, pp. 653-660.

[Hopk77] Hopkins, A.L. Jr., "Design Foundation for Survivable Integrated On-Board Computation and Control," *Proceedings of the Joint Automatic Control Conference*, 1977, pp. 232-237.

[HoSa84] Horowitz, E. and Sahni, S., *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, MD, 1984.

[HoSL78] Hopkins, A. Smith, B., Lala, J., "FTMP-A Highly Reliable Fault Tolerant Multiprocessor for Aircraft," *Proceedings of the IEEE*, Vol. 66, No. 10,

October 1978, pp. 1221-1239.

[HuAb84] Huang, K. and Abraham, J., "Fault-Tolerant Algorithms and Their Application to Solving Laplace Equations," *Proceedings of the 1984 International Conference on Parallel Processing*, August 1984, pp. 117-122.

[HuAb87] Hua, K. and Abraham, J., "Design and Evaluation of Executable Assertions for Concurrent Error Detection," *Proceedings of the 11th International COMPSAC*, Tokyo, Japan, October 1987, pp. 324-330.

[Huan83] Huang, K., *Fault-Tolerant Algorithms for Multiple Processor Systems*, Ph.D. Thesis, Coordinated Science Laboratory, University of Illinois, November, 1983.

[HuZu83] Hummel, R. and Zucker, S., "On the Foundations of Relaxation Labeling Processes," *PAMI*, Vol. PAMI-5, No. 3, May 1984, pp. 267-287.

[Hwan87] Hwang, K., "Advanced parallel processing with supercomputer architectures," *Proc. of the IEEE*, October 1987, pp. 1348-1378.

[KrTo81] Kraft, G. and Toy, W., *Microprogrammed Control and Reliable Design of Small Computers*, Prentice-Hall, 1981.

[Kueh69] Kuehn, R. "Computer Redundancy: Design, Performance, and Future," *IEEE Transactions on Reliability*, Vol. R-18, No. 1, February, 1969, pp. 3-11.

[KuLM87] Kuo, C., Levy, B. and Musicus, B., "A Local Relaxation Method for Solving Elliptic PDEs on Mesh-Connected Arrays," *SIAM Journal of Scientific and Statistical Computing*, Vol. 8, No. 4, July 1987, pp. 550-573.

[KuRe81] Kuhl, J. and Reddy, S., "Fault-Diagnosis in Fully Distributed Systems," Proc. 11th Fault Tolerant Computing Symposium, 1981, pp. 100-105.

[KuRe86] Kuhl, J. and Reddy, S., "Fault Tolerance Considerations in Large, Multiple Processor Systems," *IEEE Computer*, March 1986, pp. 56-67.

[Lala86] Lala, J., "A Byzantine Resilient Fault Tolerant Computer for Nuclear Power Plant Applications," *16th Symposium on Fault Tolerant Computing Systems*,, Vienna, Austria, July 1986, pp. 338-343.

[LaSP82] Lamport, L., Shostak, R., Pease, M., "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol 4., No. 3, July 1982, pp. 382-401.

[MaMa78] Mallela, S. and Masson, G., "Diagnosable Systems for Intermittent Faults," *IEEE Trans. Comp.*, Vol. C-27, No. 6, June, 1978 pp. 560-566.

[McCl85] McCluskey, E., "Built-in Self-test Techniques," *IEEE Design and Test*, April 1985, pp. 21-28.

[McNi87a] McMillin, B. and Ni, L., "Reliable Parallel Elliptic PDE Solution," Presented at the 3rd. SIAM Conf. on Parallel Processing for Scientific Computing, Los Angeles, CA, December, 1987.

[McNi87b] McMillin, B. and Ni, L., "Byzantine Fault-Tolerance through Application Oriented Specification," *Proceedings of the 11th International COMPSAC*, Tokyo, Japan, October 1987, pp. 347-353.

[McNi88a] McMillin, B. and Ni, L., "A Reliable Parallel Algorithm for Relaxation Labeling," *Proceedings of the 1988 International Conference on Parallel Processing for Computer Vision and Display*, Leeds, U.K., January, 1988, to appear.

[McNi88b] McMillin, B. and Ni, L., "Executable Assertion Development for the Distributed Parallel Environment," *Proceedings of the 12th International COMPSAC*, Chicago, IL, October 1988, to appear.

[McNi88c] McMillin, B. and Ni, L., "Reliable Parallel Sorting Through the Application Oriented Fault-Tolerance Paradigm," Submitted for Publication.

[McEN88] McMillin, B., Esfahanian, A. and Ni, L., "Limitations of System Level Diagnosis by Fault Model Considerations," *Proceedings of the 3rd International Conference on Supercomputing*, Boston, MA, May, 1988, to appear.

[MoHZ83] Mohammed, J., Hummel, R. and Zucker, S., "A Gradient Projection Algorithm for Relaxation Methods," *PAMI*, Vol. PAMI-5, No. 3, May 1983, pp. 330-332.

[Perr85] Perry, K., "Randomized Byzantine Agreement," *IEEE Transactions on Software Engineering*, Vol SE-11, No. 6, June 1985 pp. 539-546.

[PeSL80] Pease, M., Shostak, R. and Lamport, L., "Reaching Agreement in the presence of faults," *Journal of the ACM*, Vol 27, No. 2,, April 1980, pp. 228-234.

[PrMC67] Preparata, F., Metze, G. and Chien, R., "On the connection assignment problem of diagnosable systems," *IEEE Trans. Electron. Comput.*, vol. EC-16, December 1967, pp. 848-854.

[Quin87] Quinn, M., *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, 1987.

[Rabi83] Rabin, M., "Randomized Byzantine Generals," *Proceedings of the 24th Symposium on the Foundation of Computer Science*, Tucson, AZ, November, 1983, pp. 403-409.

[Rand75] Randall, B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 220-232.

[Renn84] Rennels, D., "Fault-Tolerant Computing-Concepts and Examples," *IEEE Transactions on Computers*, Vol C-33, No. 12, , Dec. 1984, pp. 1116-1129.

[Sham79] Shamir, A., "How to share a secret," *CACM*, Vol 22, 1979, pp. 612-613

[ShFi88] Shih, Y. and Fier, J., "Hypercube Systems and Key Applications," *Parallel Processing for Supercomputing and Artificial Intelligence,* K. Hwang and D. DeGroot, (Eds.), McGraw-Hill, New York, 1988.

[ShRa87] Shin, K. and Ramanathan, P., "Diagnosis of Processors with Byzantine Faults in a Distributed Computing System," *17th Symposium on Fault Tolerant Computing,* Pittsburg PA, July 1987, pp. 55-60.

[Skla76] Sklaroff, J., "Redundancy Management Techniques for Space Shuttle Computers," *IBM Journal of Research and Development,* vol. 20, January 1976, pp. 20-28.

[Siew82] Sieworek, D., *The Theory and Practice of Reliable System Design*, Digital Press, Bedford, MA, 1982.

[Smit65] Smith, G., *Numerical Solution of Partial Differential Equations*, Oxford University Press, New York, p. 150.

[Snyd82] Snyder, L., "Parallel Programming and the Poker Programming Environment," *IEEE Computer*, Vol. 17 No. 7, July, 1984, pp. 27-36.

[Somm82] Sommerville, I., *Software Engineering*, Addison-Wesley, London, 1982.

[Stoc87] Stockman, G. C., Private Communication, 1987.

[Stro86] Strong, R., "Problems in Maintaining Agreement," *Fifth Symposium on reliability in Distributed Software and Database Systems*, IEEE Computer Society, January 1986, Los Angeles, CA, pp 20-27.

[Stuc77] Stucki, L. "New directions in automated tools for improving software quality," *Current Trends in Programming Methodology*, Vol. 2., R.T. Yeh (Ed.), Prentice-Hall, Englewood Cliffs, N.J., 1977, pp. 80-111.

[Toyw78] Toy, Wing, "Fault-Tolerant Design of Local ESS Processors," *Proceedings of the IEEE*, Vol. 66, , October, 1978, pp. 1126-1145.

[Varg62] Varga, R., *Matrix Iterative Analysis,* Prentice-Hall, Englewood Cliffs, New Jersey, 1962.

[Wens78] Wensley, J., et. al. "SIFT: The design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE,* Vol. 66, October

1978, pp. 1240-1255.

[YaHa84] Yanney, R. and Hayes, J., "Distributed Recovery in Fault Tolerance Multiprocessor Networks," *4th International Conference on Distributed Computing Systems*, IEEE 1984, pp. 514-525.

[YaMa86a] Yang, C., Masson, G., "A fault Identification Algorithm for t_i-Diagnosable Systems," *IEEE Trans. Comp.*, Vol. C-35, No. 6, June 1986, pp. 503-510.

[YaMa86b] Yang, C., Masson, G., "A strategy for Fault Diagnosis in Asynchronous Distributed Systems with Soft Failures," *Proc. Intl. Conf. on Computer Design: VLSI in Computers*, 1986 Port Chester, NY, pp. 366-369.

[Zout76] Zoutendijk, G., *Mathematical Programming Methods*, North-Holland, Amsterdam, 1976.