EXPERIMENTAL EVALUATION OF TECHNIQUES FOR FAULT TOLERANCE

Luiz A. Laranjeira, Miroslaw Malek, and Roy Jenevein

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-92-32 July 1992

EXPERIMENTAL EVALUATION OF TECHNIQUES FOR FAULT TOLERANCE*

Luiz A. Laranjeira†
Miroslaw Malek‡ Roy Jenevein*§

Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX

Department of Computer Sciences*
The University of Texas at Austin
Austin, TX

*This research was supported in part by CAPES (Coordenacao de Aperfeicoamento de Pessoal de Ensino Superior - Brazil) under fellowship 7099/86-2, ONR under Grant N00014-88-K-0543 and NASA under Grant NAG9-426.

†Phone: (512) 471-1658, E-mail: luiz@emx.utexas.edu
‡Phone: (512) 471-5704, E-mail: malek@emx.utexas.edu
§Phone: (512) 471-9722, E-mail: jenevein@cs.utexas.edu

Abstract

In the design of critical applications that must deliver continuous service under real-time constraints it is of essence to have fault tolerance achieved with as low time redundancy as possible. Furthermore, in the case of embedded systems, space may also be a limited resource. Therefore, for critical applications, we are left with the task of providing reliable computing with low space *and* time redundancy. The quantification of space and time redundancy in the implementation of fault tolerance is consequently a must.

We present the results of practical experiments, implemented on a multiprocessor platform, with several general and application-specific technique for fault tolerance. The time and space overheads incurred by each technique are analyzed and compared. Three iterative algorithms were utilized in the experiments: solution of Laplace equations, the calculation of the invariant distribution of Markov chains, and the solution of systems of linear equations. Fault-tolerant versions of those algorithms were implemented with two general techniques for fault tolerance (triplication with voting and checkpointing and rollback) and three application-specific techniques for fault tolerance (self stabilization, algorithm-based fault tolerance, and natural redundancy). The results of the experiments show that the approach based on natural redundancy, for applications possessing that property, presents the most attractive cost/benefit ratio when only single faults are likely to occur. The implementations of the above-mentioned algorithms with this technique demanded less than 15% time redundancy for problems of significant size in fault-free situations, only one extra iteration to recover from a fault, and no extra processors. These capabilities seem ideal for critical applications that must deliver reliable service in a timely manner. Surprisingly, time overheads for some other methods were much higher than expected.

Key words: critical applications, experimental results, fault-tolerant algorithms, techniques for fault tolerance, natural redundancy, parallel algorithms, real-time computing, space/time redundancy.

1 Introduction

The cost of fault tolerance can be expressed in terms of the redundancy a computer system must have in order to tolerate faults. This redundancy is necessary in the execution of the basic tasks that represent the essence of fault tolerant-computing: fault detection, fault location and fault recovery.

In order to be fault tolerant a system may incur space and/or time redundancy (see Fig. 1a). Space redundancy is mainly related to providing an extended system state with extra information to be used for detecting, locating and recovering from faults. Space redundancy can be expressed in terms of extra variables, extra code and extra memory, extra processes and processors.

On the other hand, time redundancy is related to the time necessary to generate this extra information and to the time necessary to use this information in the execution of fault detection, location and recovery. This time overhead is crucial in the design of real-time (time critical) systems.

Also, regarding fault tolerance in multiprocessor architectures, the use of extra processors in the execution of an application may imply less time redundancy since the procedures that implement fault tolerance could be executed in parallel in less time. Conversely, the straight use of time redundancy (e.g, recomputation) would certainly make the utilization of extra processors unnecessary.

Since space and time redundancy are intertwined in the fault tolerance process, a fundamental tradeoff in the design of reliable systems can be observed. Generally, for a given technique for fault tolerance, the more space redundancy it requires, the less time redundancy it will need in order to ensure dependable execution of applications, and vice versa (see Fig. 1b).

In the design of critical applications that must deliver continuous service under real-time constraints it is of essence to have fault tolerance achieved with as low time redundancy as possible. Furthermore, in the case of embedded systems, space may also be a limited resource. Therefore, for critical applications, we are left with the task of providing reliable computing with low space *and* time redundancy. This situation could be represented by a point such as P_o in Fig. 1b. The quantification of time and space redundancy in the implementation of fault tolerance is consequently a must.

Several techniques for fault tolerance have been proposed in the literature. Some are general, others are application-specific. These techniques appear to be based on sound principles which are presented in many research papers together with theoretical analyses of the expected space and time overheads they will cause. There is, however, little experimental data reflecting this cost in actual implementations of these techniques.

In this paper we intended to partially fill this gap by presenting the results of experiments we conducted implementing three algorithms in a multiprocessor platform using different techniques for fault tolerance. We present measurements of the space and time redundancy required for each technique. Concerning time redundancy, we look at redundancy in fault-free situations and redundancy necessary to execute recovery when a fault happens. The fault coverage of the utilized fault detection procedures are also presented.

The algorithms we utilized in our experiments were: solution of Laplace equations by Jacobi's method, solution of systems of linear equations also by Jacobi's method, and the computation of the invariant distribution of Markov chains.

We analyze general techniques for fault tolerance such as triplication with voting and checkpointing and rollback, and application-specific techniques such as algorithm-based fault tolerance, self stabilization, and natural redundancy.

The results of the experiments point out that application-specific techniques are more likely to provide the low space and time overheads required by critical applications. More specifically, for situations where only single (either temporary or permanent) faults are likely to occur, the approach based on natural redundancy, provided that property is present in the application, offers the most attractive cost/benefit ratio. This technique requires no extra processor and very low time redundancy in fault-free situations as well as when a fault occurs. The implementation of the Laplace algorithm with this technique caused a time overhead of less than 15% in fault-free situations for problems of significant size. The Markov algorithm implementation incurred less than 8% time overhead in fault-free situations for all data sets utilized, while the linear equations algorithm implementation incurred less than 10% time overhead in the same conditions for all data sets utilized except one (a 60-variables problem, executed with 10 processors, for which a 10.72% overhead was observed). In all cases one extra iteration was necessary to execute fault recovery. An obvious drawback of this approach is that it applies only to algorithms possessing natural redundancy.

This paper is divided as follows. In Section 1 we describe the model of computation we utilized. Section

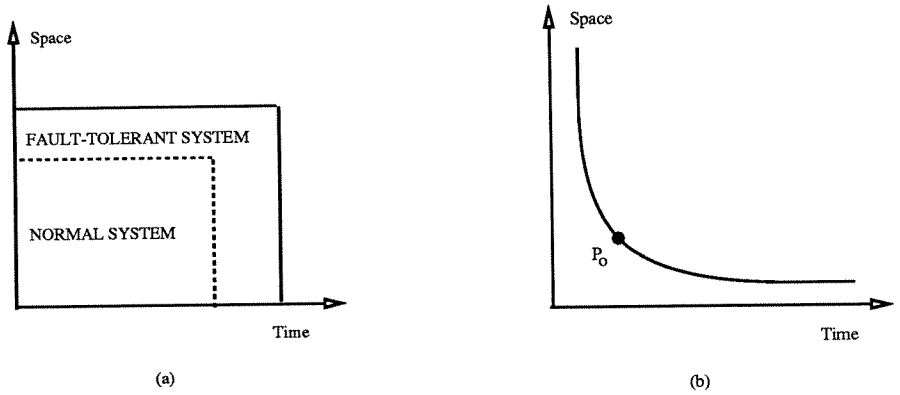


Figure 1: (a) Time and space redundancy needed for fault-tolerant system implementation. (b) Tradeoff between space and time redundancy required for fault tolerance.

2 presents the target architecture and the assumed fault model. Section 3 contains a brief overview of the techniques we experimented with. Section 4 includes a short description of each algorithm and some implementation details of the schemes for fault tolerance. In Section 5 a discussion of the testbed and experimental results quantifying space/time overhead are presented. Finally concluding remarks are in Section 6.

2 The Parallel Model of Computation

In this work we have adopted a model of computation which is based on the bulk-synchronous model of parallel computation proposed by Valiant [1] with some convenient extensions and restrictions. In this model the execution of a parallel algorithm proceeds in *supersteps*. Only one process runs in each processor. So we will use the words process and processor interchangeably. The processes participating in a superstep are initially given a *step* of L time units to execute a given amount of processing. After each period of L time units, a global check is made in order to determine if the superstep has been completed by all participating processes. If that is the case, the computation advances to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep. The model assumes the existence of facilities for a barrier synchronization of processes at regular intervals of L time units where L is the *periodicity parameter*. The value of L may be controlled by the program, even at runtime. This synchronization mechanism captures in a simple way the idea of global synchronization at a controllable level of coarseness. The realization of such a mechanism in hardware would provide an efficient way of implementing tightly synchronized parallel algorithms without overburdening the programmer.

The execution of a superstep can be seen as a global function that maps the current state to the next state of a computation. Considering that the state of a computation is composed by the values of certain variables, the code in each process can be viewed as an update function for a portion of the total state of the computation. We consider that a process may or not be enabled and that only enabled processes are executed at a given superstep. Each process has its own enabling condition, which could be true, for instance, if a certain predicate holds in the current state of the computation. All enabled processes run in parallel at a given superstep.

We place a restriction on Valiant's model by requiring information to be exchanged among processes only at the end of a superstep. This information exchange can be accomplished either by shared memory access or by message passing and is considered to be part of the superstep execution. The system state is undefined during the execution of a superstep. Philosophically, this is the same as viewing a superstep (considering the information exchange as part of it) as a state transition. Therefore, the state of the system is defined only in between supersteps. The reason for limiting information exchange to happen only at the end of a superstep is to guarantee that processing or communication faults occurring during a superstep execution do not contaminate the entire computation. By executing a correctness check (fault detection) in between supersteps this contamination can be avoided.

Before the execution of a superstep a check is made in order to determine which processes are enabled at

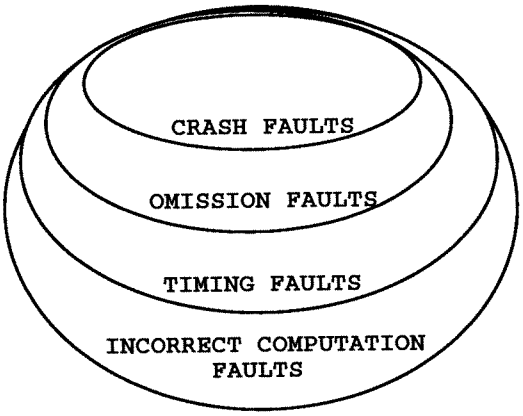


Figure 2: A nested fault classification scheme.

the current state. We do not concern ourselves whether this check is implemented in a local or global fashion. It is assumed, however, that the check is realized in finite time.

In our model a computation is terminated after reaching a certain state in which there are no enabled processes. The enabling condition of each process should be designed in such a way that it can capture this termination criterion by evaluating to FALSE when the objective of the computation is reached.

3 Target Architecture and Fault Model

We will consider as our target architecture a time-bounded, asynchronous, shared memory MIMD machine such as the Sequent Symmetry, where a number of processors are linked by a common bus. The parallelism in such an architecture is considered to be a coarse-grained parallelism.

We adopt a fault classification scheme, similar to the one in [2], where several nested fault classes are considered (see Fig. 2). We consider that processors operate by responding to triggering events such as the acquisition of a lock or the reaching of a synchronization point. A crash fault occurs when a processor systematically stops responding to its triggering events. An omission fault occurs when a processor either systematically or occasionally does not respond to a triggering event. A timing fault occurs when, in response to a triggering event, a processor gives the right output too early, too late or never. An incorrect computation fault may be a timing fault or a computation which is delivered on time but contains wrong or corrupted output values. Since this is a nested fault model, incorrect computation faults include timing faults; timing faults include omission faults; and omission faults include crash faults. Another aspect of our fault model is that faults may be permanent or temporary. Crash faults are always permanent, whereas omission, timing and incorrect computation faults can be either permanent or temporary.

Since our model of computation indicates that algorithms are executed in supersteps, we allow one processor per superstep to produce erroneous results due to temporary (transient or intermittent) faults or one processor under a permanent faulty condition.

We view the applications as fault-free, that is, software design faults are not considered. We assume that the bus is fault-free and memory faults are detected and corrected by error detection/correction codes, such as Hamming codes. We also consider that the address generation logic of processors and the address decoding circuits of the memory system are reliable. This can be achieved by hardware redundancy schemes such as self-checking logic or replicated logic with voting (see [3]).

Our assumptions are consistent with the view that fault tolerance should be implemented at every layer of a computational system, allowing different types of faults to be tolerated at different levels, in order to optimize resources and minimize performance overhead. In this work we consider tolerating faults in the computation paths.

4 Description of the Techniques for Fault Tolerance

In this section we provide a brief description of each of the techniques for fault tolerance we used in our experiments. The techniques we cover are: (i) General techniques such as triplication with voting [3] and checkpointing and rollback [4]; (ii) Application-specific techniques such as algorithm-based fault tolerance [5], self stabilization [6], and natural redundancy [7]. A high level view of the application-specific techniques shows that self stabilizing algorithms do not need explicit procedures to implement fault tolerance, naturally redundant algorithms posses inherent redundancy which still needs to be exploited by the implementation of explicit procedures, and algorithm-based fault tolerance is viewed as a technique which explicitly inserts "artificial" redundancy and the procedures to exploit it to the algorithm. We encourage the interested reader to refer to the references for more details on these techniques.

Triplication with Voting:

In this technique processors are triplicated. At specific points of the computation the partial results computed by the processors in each set of sibling processors are compared and a majority vote is taken to choose the correct result. The voting procedure takes care of both fault detection and recovery. The technique works as long as a majority of processors (or communication hardware) in each triplicated set is not simultaneously faulty. Both temporary and permanent faults can be tolerated by using triplication with voting.

Checkpointing and Rollback:

In this technique a checkpoint of a correct state is periodically stored in stable storage. Upon the detection of a fault the last state stored as a checkpoint is restored and the execution of the application restarts from that state. This technique is normally used to tolerate temporary faults. In order for this technique to work it is necessary to implement some kind of fault detection procedure, but no fault location is necessary. An important parameter in the implementation of this technique is the checkpointing interval I. In general, papers describing this approach do not address the issue of fault detection. It is normally assumed that faults can be detected but the specific detection scheme is not described. In our implementations of checkpointing and rollback we applied the fault detection approach that was most suitable for each algorithm.

Algorithm-Based Fault Tolerance:

In this technique, mainly used with problems involving matrix operations, the calculation of checksums of the output variables are added to the problem in order to allow fault tolerance to be achieved. Faults are identified when a checksum does not match the actual sum of output variables. Normally a simple and a weighted checksum are necessary in order that fault detection, location and recovery might be achieved. The simple checksum corresponds to a mere addition of the output variables, while in the case of the weighted checksum the addition is performed assigning weights to the output variables (see [5]). The checksums must be computed independently from the normal output results. This may imply the utilization of extra processors to compute them in parallel with the normal application in order to ensure small time overhead in fault-free situations. If the fault detection procedures and the computation of an extended state for recovery is added to the normal computations and spread over the original set of processors this may cause high time overhead. Although other methods (based on the characteristics of the application) for inserting redundancy could be used, by far the most common utilization of the algorithm-based approach for fault tolerance is the implementation of checksums. Algorithm-based fault tolerance has been proposed to tolerate temporary single faults. This technique will only work with algorithms for which checksums of the output values (or other means for inserting redundancy) can be calculated with small time overhead with respect to the calculation of the normal output values.

Self Stabilization:

Self-stabilizing algorithms were first noticed by Dijkstra [6] as algorithms that can tolerate temporary faults with no need of adding explicit procedures for fault tolerance. In the execution of these algorithms a state plagued by a temporary fault is still a valid initial state. It is assumed that as a computation continues from a

faulty state a fault-free state will eventually be reached. This technique will only work for algorithms possessing the desirable self-stabilization properties.

Natural Redundancy:

Naturally redundant algorithms, proposed in [7], are algorithms that possess implicit redundancy in the problem variables. This already existing redundancy can be used for fault recovery and sometimes for fault detection also. Possessing redundancy, however, does not necessarily imply automatic recovery from faults. In order to restore a faulty state, an explicit recovery procedure must still be added to the algorithm. Naturally redundant algorithms are in an intermediary position between algorithms that need neither redundancy nor recovery to be added to them and algorithms that, in order to be made fault tolerant, need both redundancy and recovery to be inserted in their executions. This technique for fault tolerance allows algorithms to tolerate both temporary and permanent single faults, but it is only applicable to algorithms possessing the desirable natural redundancy properties.

5 Description of the Algorithms

In this section we provide a brief description of the algorithms we used in our experiments together with a concise explanation on how the different techniques for fault tolerance were implemented for each of them. It should be noticed that the three algorithms we experimented with are naturally redundant. As such one would not need to use other techniques that insert redundancy in order to make these algorithms fault tolerant. The implementation of these algorithms with other techniques was done for the sake of comparing the efficiency of the various techniques with the same algorithm. A general strategy for designing fault-tolerant algorithms would be to look for natural redundancy or self stabilization properties first. If those properties are not present or if the fault toleranc they provide does not meet system specifications (tolerable time and/or space overhead, expected fault coverage) then one would need to explicitly insert redundancy utilizing other techniques such as replication, checkpointing or algorithm-based fault tolerance.

5.1 Solution of Laplace Equations

Description of the Algorithm

The solution of Laplace equations is required in the study of important problems such as seismic modeling, weather forecasting, heat transfer and electrical potential distributions. Equation 1 is a two-dimensional Laplace Equation.

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0 \tag{1}$$

The usual approach to an iterative solution consists of discretizing the problem domain with an n x n uniformly-spaced square grid of points, so that all n^2 points, except those on boundaries, have four equidistant nearest neighbors. Equation 1 is then approximated by a first order difference equation such as Equation 2, where x and y are the row and column indices over all grid points, and the value of function ϕ at each grid point is calculated in an iterative fashion until convergence is achieved.

$$4\phi_{x,y} - \phi_{x-1,y} - \phi_{x+1,y} - \phi_{x,y-1} - \phi_{x,y+1} = 0 \tag{2}$$

On of the most common iterative techniques used to solve Laplace equations is Jacobi's method. Equation 3 shows the update procedure for Jacobi's method. In order to calculate the next iteration value of a point one needs the values of its nearest neighbors calculated at the previous iteration.

$$\phi_{x,y}^{k+1} = \frac{\phi_{x-1,y}^{k} + \phi_{x+1,y}^{k} + \phi_{x,y-1}^{k} + \phi_{x,y+1}^{k}}{4} \tag{3}$$

Let us consider Φ^k the vector composed by the values of all grid points $\phi^k_{x,y}$ after the k^{th} iteration. In general, the sequence defined by $\{\Phi^k\}$, $k \geq 0$, will converge to a solution $\Phi^* = (\phi^*_{1,1}, \phi^*_{1,2}, ..., \phi^*_{n,n})$. In practice, however, one cannot obtain the exact final solution Φ^* due to computer finite wordlength limitations. A convergence criterion is then established which is defined by an approximation factor ϵ. The execution of the algorithm should stop after the k^{th} iteration if $|\phi^k_{x,y} - \phi^*_{x,y}| \leq \epsilon, 1 \leq x, y \leq n$. As the values of the $\phi^*_{,xy}$ are not known in most cases, convergence is considered to be achieved if $|\phi^k_{x,y} - \phi^{k-1}_{x,y}| \leq \epsilon, 1 \leq x, y \leq n$.

This algorithm is implemented on a p processor MIMD architecture by dividing the grid of points in p subgrids and assigning each subgrid to a processor. Processors must then exchange the values of the points on the boundaries of the subgrids in order to calculate the next iteration of points. Each iteration corresponds to a superstep in the assumed model of computation.

Fault-Tolerant Implementations

The implementation of triplication with voting is straightforward. By the end of each superstep each set of triplicated processors compares its computed values and performs a majority vote in case there is any discrepancy. This comparison, however, is done upon a row or column of points of each subgrid (not upon all points of the subgrid) in order to avoid excessive performance overhead.

The implementation of checkpointing and rollback is also straightforward. Fault detection is accomplished by having each processor calculating an extra row or column belonging to a neighboring processor and comparing those values with the ones calculated by the neighboring processor.

The implementation of the algorithm-based approach was not done for this algorithm. The processor calculating the checksum would do at least twice as much work as a processor updating one of the subgrids of points. The reason for this is that there is no inexpensive way of calculating the checksum other than calculating the values for each processor (each subgrid) and adding them up. One could save some work by dividing by four only once in this calculation, but the work corresponding to the additions cannot be avoided (see Equation 3). The higher the number of processors used in the computation, the higher would it cost to calculate the checksum. This would correspond to a very high time overhead in fault-free situations raising serious doubts about the practicality of such implementation.

This algorithm is self-stabilizing. Since this is a property that is inherent to the algorithm, no rework needed to be superimposed to the normal version of the algorithm in order to achieve it. No explicit mechanisms for fault detection, location or recovery are necessary here.

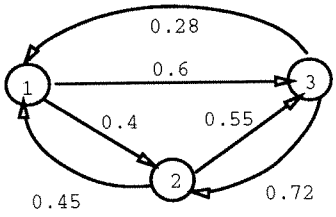
This algorithm is also naturally redundant. Basically the natural redundancy comes from the fact that the grid points can be divided in odd and even points and that the next iteration of even points can be computed using only odd points (and vice versa). An odd (even) point is one for which the sum of its row and column indices is an odd (even) value. Two clusters of processors are used. One cluster calculates the odd points and the other calculates the even points at each iteration. If a processor on one of the clusters fails when calculating odd (even) points, then the even (odd) points computed by the corresponding processor on the other cluster can be used for recovery. Fault detection is accomplished using the same procedure utilized with the checkpointing and rollback technique. More details on this implementation can be found in [7]. For fault location (diagnosis), we used a distributed algorithm similar to the one in [8]. Due to the Sequent Symmetry architecture (shared memory) and our assumption of reliable memory accesses, each processor can correctly access the test results of every other processor. This way all the fault-free processors can correctly diagnose a faulty processor without needing several communication rounds as would be necessary in a message passing architecture. In our implementation, the test results already available from the fault detection procedure were utilized in the diagnosis algorithm.

Together with the fault detection procedures described for each version of the algorithm (except the self-stabilizing version) timeouts may be utilized to detect crash, omission and timing faults where applicable.

5.2 Computation of the Invariant Distribution of Markov Chains

Description of the Algorithm

The Markov process model is a powerful tool for analyzing complex probabilistic systems such as those used in queueing theory and computer systems reliability and availability modeling. The main concepts of this model



(a)

$$\begin{bmatrix} 0.0 & 0.4 & 0.6 \\ 0.45 & 0.0 & 0.55 \\ 0.28 & 0.72 & 0.0 \end{bmatrix}$$

(b)

Figure 3: (a) Graph representation of a three-state Markov model. (b) The corresponding transition probability matrix T.

are state and state transition. As time goes by, the system goes from state to state under the basic assumption that the probability of a given state transition depends only on the current state. We are particularly interested here in the discrete-time time-invariant Markov model, which requires all state transitions to occur at fixed time intervals and transition probabilities not to change over time.

Figure 3(a) shows a graph representation of a three-state Markov model. The nodes represent the states of the modeled system, the directed arcs represent the possible state transitions and the arc weights represent the transition probabilities. The information conveyed in the graph model can be summarized in a square matrix T (Fig. 3(b)), whose elements t_{ij} are the probabilities of a transition of a state i to a state j in a given time step. Such an n x n square matrix T is called the transition probability matrix of an n-state Markov model. T is a *stochastic* matrix, since it meets the following properties: $t_{ij} \geq 0$ for $1 \leq i, j \leq n$, and $\sum_{j=1}^{n} t_{ij} = 1$ for $1 \leq i \leq n$.

A discrete-time, finite state Markov chain is a sequence $\{X^k \mid k = 0, 1, 2, ...\}$ of random variables that take values in a finite set (state space) $\{1, 2, ..., n\}$ and such that $\Pr(X^{k+1} = j \mid X^k = i) = q_{ij}$, $k \geq 0$, where Pr means probability. A Markov chain could be interpreted as the sequence of states of a system modeled by a Markov model with the probabilities q_{ij} given by the entries t_{ij} of the transition probability matrix T.

Let Π^0 be an n-dimensional nonnegative row vector whose entries sum to 1. Such a vector defines a probability distribution for the initial state X^0 by means of the formula $\Pr(X^0 = i) = \pi_i^0$. Given an initial probability distribution Π^0, the probability distribution Π^k, corresponding to the k^{th} state X^k, would be given by Equation 4, where T^k means T to the k^{th} power.

$$\Pi^k = \Pi^0 T^k \qquad k \geq 0 \tag{4}$$

Equivalently,

$$\Pi^{k+1} = \Pi^k T \qquad k \geq 0 \tag{5}$$

It is often desired to compute the steady-state (invariant) probability distribution Π^{ss} for a Markov chain. The vector Π^{ss} is a nonnegative row vector whose components sum 1, and has the property $\Pi^{ss} = \Pi^{ss} T$.

It can be shown that the iteration given by Equation 5 converges to Π^{ss} if the initial vector Π^0 is such that $\sum_{i=1}^{n} \pi_i^0 = 1$ (this is always the case when the values of the components of vector Π represent probabilities). Assuming fault-free execution, after each iteration the sum of the values of the components of Π will be always equal to 1. The interested reader can find the proof for those propositions in [9].

Normally the value of n is very large as compared to the number of processors p available on a MIMD architecture. The solution for this problem is then implemented by having each processor updating n/p components of Π at each iteration (superstep). A convergence criteria would be $\{\forall\ i,\ 1 \leq i \leq n \mid (\pi_i^k - \pi_i^{k-1}) \leq \epsilon\}$, for some given ϵ.

Fault-Tolerant Implementations

The implementation of triplication with voting is straightforward. By the end of each superstep each set of triplicated processors compares its computed values and performs a majority vote in case there is any discrepancy.

The implementation of checkpointing and rollback is also straightforward. Fault detection is accomplished after each superstep by checking if the sum of the components of the output vector Π add up to 1.

For the algorithm-based fault-tolerant version of this algorithm we implemented a cross-checksum in order to make fault recovery possible. Fault detection is accomplished by the same procedure used in the checkpointing implementation. An additional processor calculates a vector of checksums. Assuming each processor calculates n/p components of the output vector Π in each iteration, the vector of checksums will also have n/p components. The l^{th} component of this vector is the checksum of all l^{th} components of the vectors calculated by each processor. When processor i is faulty during a certain iteration, correct values for that processor can be recovered using the values calculated by the other processors and the cross-checksums calculated by the additional processor. For fault location we used the same procedure utilized with the naturally redundant version of the Laplace algorithm.

This algorithm does not have self-stabilization properties, therefore a self-stabilizing version of it was not implemented.

This algorithm is also naturally redundant. Basically the natural redundancy comes from the fact that the sum of the components of vector the output vector Π is equal to 1 after each iteration (superstep). This property can be used to detect faults and to recover from single faults, after fault location is accomplished by another method. We perform Jacobi-like iterations. When a fault plagues processor i during iteration k the correct values calculated by that processor in iteration $k-1$ (which are still available in the machine's memory) can be used, together with the property that $\sum_{i=1}^{n} \pi_i^k = 1$, in order to recover acceptable values corresponding to the ones that were contaminated by the fault. The same fault location procedure used with the algorithm-based version is used here also. The values of the tests for fault location, however, are calculated by an approach similar to RESO [10]. The difference is that we do a shifted recomputation at the processor level, not at the register level. If a fault is detected by the fault detection procedure in the k^{th} iteration, the fault diagnosis procedure is started and processor i tests processor $(i+1)$ by recalculating the value of π_{i+1}^k and comparing it to the value previously calculated by that processor (if there are p processors, processor p tests processor 1).

Together with the fault detection procedures described for each version of the algorithm timeouts may be utilized to detect crash, omission and timing faults where applicable.

5.3 Solution of Systems of Linear Equations

Description of the Algorithm

Systems of linear equations often appear as part of scientific and engineering problems. Usually we have n unknown variables and n equations in a system represented as a matrix equation such as:

$$A * X = B \tag{6}$$

Which can be expanded as:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & a_{n3} & \dots & a_{nn} \end{vmatrix} * \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{vmatrix} = \begin{vmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{vmatrix} \tag{7}$$

An iterative solution for this problem was proposed by Jacobi. Given an arbitrary initial value for X, say X^0, it consists of calculating successive values for X, using the recurrence relation given by Equation 8, until a convergence criteria is met.

$$X^{k+1} = -D * X^k + E \tag{8}$$

Matrices D and E of Equation 3 are obtained from matrices A and B of Equations 6 and 7 in the following way: (a) $d_{ii} = 0$, $1 \leq i \leq n$; (b) $d_{ij} = a_{ij}/a_{ii}$, $i \neq j$, $1 \leq i, j \leq N$; (c) $e_i = b_i/a_{ii}$, $1 \leq i \leq n$.

Normally the value of n is very large as compared to the number of processors p available on a MIMD architecture. The solution for such problem is then implemented by having each processor updating n/p components

of X at each iteration (superstep). A convergence criteria would be $\{\forall\ i,\ 1 \leq i \leq n \mid (x_i^k - x_i^{k-1}) \leq \epsilon\}$, for some given ϵ.

Fault-Tolerant Implementations

The implementation of triplication with voting is straightforward. By the end of each superstep each set of triplicated processors compares its computed values and performs a majority vote in case there is any discrepancy.

The implementation of checkpointing and rollback is also straightforward. Fault detection is accomplished by letting each processor calculate a checksum of the values calculated by another processor (the tested processor). After each superstep each processor checks if the checksum it has calculated is equal to the sum of the components of vector X calculated by the tested processor.

For the algorithm-based fault-tolerant version of this algorithm we utilized the same fault detection mechanism as used for the checkpointing and rollback implementation. The fault recovery and location procedures were the same as those implemented with the algorithm-based version of the Markov algorithm.

This algorithm is self-stabilizing. Since this is a property that is inherent to the algorithm, no rework needed to be superimposed to the normal version of the algorithm in order to achieve it. No explicit mechanisms for fault detection, location or recovery are necessary here.

This algorithm is also naturally redundant. Basically the natural redundancy comes from the fact that in order to calculate the value of one component of vector X in the k^{th} iteration one needs only the values of the other components calculated in iteration $k-1$. So, if the i^{th} processor is faulty at iteration k, its component values can be recovered, one by one, with the normal iteration procedure as the recovery procedure, and utilizing the values of the components of X calculated by the other processors at iteration k and the values calculated by processor i (except the one that is being recovered) in iteration $k-1$. Fault detection and fault location are accomplished in the same way as for the algorithm-based version.

Together with the fault detection procedures described for each version of the algorithm (except the self-stabilizing version) timeouts may be utilized to detect crash, omission and timing faults where applicable.

6 Experimental Results

6.1 Testbed Description and Discussion of Main Issues

We ran our experiments on a Sequent Symmetry bus-based MIMD architecture with a configuration of 12 processors. We implemented normal and fault-tolerant versions of the three algorithms with the various techniques for fault tolerance using the programming language "C" with a library of parallel extensions called PPL (Parallel Programming Language). For the sake of simplicity we will call the algorithm for solving Laplace equations by Jacobi's method as *the Laplace algorithm*, the algorithm for computing the invariant distribution of a Markov chain as *the Markov algorithm*, and the algorithm for solving systems of linear equations as *the linear equations algorithm*. We do not intend to establish a formal terminology by adopting the above mentioned names in the scope of this work. Rather our purpose is to provide short labels that may facilitate identifying and referencing each algorithm. The main issues involved in our experiments are: fault insertion, the effects of finite precision arithmetic, the error coverage and the performance overhead introduced by the schemes for fault tolerance (relative to the non fault-tolerant version of the algorithms) both for executing recovery and in fault-free situations.

In order to test the fault-tolerant implementations we utilized a scheme similar to the one described in [11], with the difference that we included fault detection, fault location, and fault recovery, while that work focused on fault detection. Hardware faults were simulated using software techniques. We used C statements to manipulate bits and words turning correct data into erroneous data at specific points in the computation paths. We used normal floating point precision. A real number is represented by a 32-bit word, including a sign bit (bit 31), 7 bits composing the exponent (bits 24 to 30), and 24 bits composing the mantissa of the floating point representation (bits 0 to 23). A bit error was introduced by flipping a bit and a word error was introduced by flipping all the bits in a word. Bit and word errors, both transient and permanet, were inserted in the calculation of mutiplications and additions during the execution of the algorithms. These inserted errors simulate faults in the adder and multiplier of the ALU (Arithmetic Logic Unit) of the processors. In the case of the Laplace algorithm, we assumed that a transient fault lasts for a period of time that is equal or greater than

the computation time needed for calculating one row or column of points (whichever is larger) in a partition of the grid. This assumption is not necessary for the Markov and linear equations algorithm. We considered a permanent fault one that lasts more than three iterations.

The effects of finite precision arithmetic can influence the error detection procedures. The error detection procedures verify whether the output data of the computations meet a certain relation or is equal to a replicated computation. In the case of comparing replicated computations for error checking (which was the approach used in all versions of the Laplace algorithm and in the triplication with voting versions of the Markov and the linear equations algorithm) finite precision arithmetic causes no problem. The reason for this is that any roundoff errors due to finite precision arithmetic will be the same in the different executions of the equivalent computations used for testing (evidently we are considering homogeneous processors). So, if an error is introduced and affects one of them the comparison will detect it.

If the error checking is done by verifying whether the computation results meet a certain relation or property, than the effects of finite arithmetic are important and need to be considered. This is the case in the implementations of the Markov and linear equations algorithms (checkpointing, algorithm-based and natural redundancy versions). In these situations the quantities that are compared are obtained through different computational paths and, thus, will have different roundoff errors. The magnitude of the roundoff error is a function of the wordlength. The shorter the wordlength, the larger the error. The norm of the computational error has been determined to be equal to $C * C(N) * 2^{-2m}$, where C is an application specific constant, $C(N)$ is a function of the problem size and also of the application, and m is the number of bits of the mantissa in the floating point representation. For fixed wordlength, problem size and application the norm of the roundoff error can be expressed as a constant δ.

Concerning the time redundancy due to the schemes for fault tolerance, the most important measure is the extra computation time in fault-free situations, which will represent most of the actual situations in which the application will run. This shows how much extra time will each superstep take in the execution of the fault-tolerant versions of the algorithms.

Other measures are the extra number of supersteps necessary to execute fault recovery, and the space redundancy incurred by each technique for fault tolerance.

6.2 Discussion of the Experiments and Comparative Analysis

At first we discuss the experiments we performed in order to evaluate the fault coverage of the fault detection schemes. For the detection schemes based on comparison of quantities obtained by equal computation paths (like the ones used in the Laplace algorithm and in the triplication with voting versions of the Markov and linear equations algorithms) a 100% coverage was observed for the faults we simulated.

Well established fault tolerance terminology states that a fault is an erroneous state of the computation, while an error is the manifestation of a fault causing the computing system to deliver incorrect results. Since our detection procedures operate by checking the validity of intermediate values of the computations, what we really can detect are errors (which are caused by faults). Since an error must always be caused by a fault, in our discussion we use the terms error checking and fault detection interchangeably.

In order to measure the fault coverage of the fault detection procedures used with the Markov and linear equations algorithms (except for the triplication with voting versions) we used four different data sets and four different sets of processors. Since here the error checking procedure compares quantities that are calculated through different data paths, roundoff errors are important. The error checking is done by subtracting the two quantities and comparing the absolute value of the result of this subtraction to a certain δ which accounts for roundoff errors. If δ is too small, *false alarms* will happen. That is, correct computations will be thought of as erroneous. If this delta is too large, many errors will not be detected. The solution we employed was to experimentally determine, for each data set, the minimum value of δ that causes no false alarms in a fault-free execution of the algorithm. This δ represents the maximum value of the finite arithmetic error for that fixed data set and problem size. Using this value for δ, we measured the fault coverage of the schemes for fault tolerance.

The percent of detected faults is given in Tables 1 and 2. It is noticeable that all faults causing word errors were covered. In particular those causing floating exceptions are detectable through watchdog timers. Most of the faults that were not detected were simulated by errors inserted in the lower order bits of the floating point representation and caused no harm to the final outcome of the algorithm. This fact plus the data shown

in the tables confirm that the fault coverage of these schemes for fault tolerance for practical considerations is quite acceptable. The percentages of fault coverage we measured in these experiments are very similar to those observed in the research on algorithm-based fault detection schemes (see [11]).

FAULT DETECTION COVERAGE - MARKOV ALGORITHM Percent of detectable and recoverable faults					$\epsilon = 1.0 * 10^{-7}$
Error type	Unit	Data set 1 $\delta = 5.74 * 10^{-7}$ 6 processors	Data set 2 $\delta = 4.04 * 10^{-7}$ 8 processors	Data set 3 $\delta = 1.12 * 10^{-7}$ 10 processors	Data set 4 $\delta = 0.73 * 10^{-7}$ 12 processors
Transient	adder	78.13	81.25	81.25	78.13
bit error	multiplier	78.13	81.25	84.38	71.86
Transient	adder	100.00	100.00	100.00	100.00
word error	multiplier	100.00	100.00	100.00	100.00
Permanent	adder	93.75	93.75	87.50	84.38
bit error	multiplier	87.50	87.50	90.63	84.38
Permanent	adder	100.00	100.00	100.00	100.00
word error	multiplier	100.00	100.00	100.00	100.00

Table 1: Fault coverage of the fault detection scheme used with the Markov algorithm (except the triplication with voting version).

FAULT DETECTION COVERAGE - LINEAR EQUATIONS ALGORITHM Percent of detectable and recoverable faults					$\epsilon = 1.0 * 10^{-5}$
Error type	Unit	Data set 1 $\delta = 3.82 * 10^{-6}$ 6 processors	Data set 2 $\delta = 7.64 * 10^{-6}$ 8 processors	Data set 3 $\delta = 1.15 * 10^{-5}$ 10 processors	Data set 4 $\delta = 1.54 * 10^{-5}$ 12 processors
Transient	adder	84.38	84.38	87.50	81.25
bit error	multiplier	75.00	75.00	68.75	75.00
Transient	adder	100.00	100.00	100.00	100.00
word error	multiplier	100.00	100.00	100.00	100.00
Permanent	adder	100.00	100.00	100.00	100.00
bit error	multiplier	100.00	100.00	100.00	100.00
Permanent	adder	100.00	100.00	100.00	100.00
word error	multiplier	100.00	100.00	100.00	100.00

Table 2: Fault coverage of the fault detection scheme used with the linear equations algorithm (except the triplication with voting version).

Now we proceed to analyze the space and time overheads observed in our experiments. It is worth mentioning again that two different time overheads were measured: the time overhead needed for generating redundancy and executing the fault detection procedures (measured by running the algorithms under a fault-free condition) and the time overhead needed for recovery when a fault does occur. These two will be analyzed separately.

In Table 3 the space redundancy is shown, in terms of the number of extra processors, and the time redundancy necessary for fault recovery is given in terms of number of extra supersteps, for the various techniques for fault tolerance we are investigating. The types of faults tolerated by each technique are also presented.

Replication with voting requires the largest amount of space overhead. Variables and processes are at least triplicated. On the other hand, the time overhead for recovery is minimal in terms of number of supersteps. If a fault occurs in one superstep, recovery is executed in the next superstep. As will be discussed later, however, the time redundancy incurred by this technique in fault-free situations is considerable for bus-based multiprocessor

architectures such as the Sequent. This technique covers a large set of faults, both temporary and permanent.

The checkpointing and rollback technique requires no extra processors. This technique is usually utilized to tolerate temporary faults. The time redundancy necessary for recovery may vary depending on how far, in terms of number of supersteps, is the superstep in which the fault occurred from the one in which the latest correct state was saved. An upper bound for this distance is I, which is the interval, in terms of number of supersteps, between two checkpoints. Previous work in this area (see [12] and [13]) report that the optimal checkpointing interval for a given system is a function of the time required to perform a checkpoint and the failure rate of the system. As failure rate information was not available for the computer system we utilized, we chose values of I for our experiments that were reasonable with respect to the problems we worked with. Since the case studies we utilized to measure time redundancy were run to completion with 500 to 1000 supersteps (iterations), we picked values of I (50, 100 and 200) that were smaller than those numbers.

Algorithm-based fault tolerance is accomplished with small space overhead. The time overhead for recovery is also minimal, basically one extra superstep is necessary. In our implementations the computation of the fault detection procedure was spread over the original set of processors, but the cross-checksums, necessary for the recovery procedure, were calculated by an extra processor in order to minimize the time overhead in fault-free situations. Otherwise (i.e., without the extra processor) this time overhead may become prohibitive.

Self stabilization requires no space redundancy. On the other hand, the time redundancy necessary for the algorithm to converge after the occurrence of a fault is not predictable and may be quite large. Tables 4 and 5 present the results of an experiment where transient bit errors were inserted during the execution of the Laplace and linear equations algorithms (which are self stabilizing) simulating faults. The number of supersteps necessary for convergence under a faulty condition, NIC_{fc}, is shown for different locations of bit errors, bit_{er}. The number of supersteps necessary for convergence in the fault-free execution of the algorithms is given by NIC_{ff}. Transient bit errors were inserted during iteration 3165 of the Laplace algorithm, and iteration 25 of the linear equations algorithm, that is, approximately when half of the entire computation had been executed.

As it is shown in these tables, the time overhead necessary for the complete execution of the algorithm plagued by one fault happening halfway of the computation can be considerable. This overhead depends on how far, in terms of the number of supersteps (iterations), the state resulting from the fault is from the state where convergence will be reached. In certain situations the inserted bit error caused a floating exception to happen (see Table 4), killing the corresponding process and causing, therefore, a permanent fault. This permanent fault could not be tolerated because self stabilization can only tolerate temporary faults.

It is also worth noticing that word errors actually happening in a real computation could include any combinations of bit errors and incur very high execution time overheads. This fact, together with the nonnegligible probability of permanent faults, seem to point out that self stabilization may not be enough to achieve the high levels of reliability and performance required by many critical applications. Another observation that can be derived from this experiment is that, since some transient faults can cause permanent faults (e.g. floating exceptions), schemes for fault tolerance that do not provide on-line recovery for permanent faults may be unrealistic for a large class of applications.

The time overhead for recovery incurred by the approach based on natural redundancy was found to be only one superstep. This technique can be used to tolerate single transient faults. It can be also easily extended to handle permanent single faults by the addition of a reconfiguration procedure and the availability of spare processors. In our experiments this extension was implemented and the algorithm versions coded with the extended technique were able to tolerate single permanent faults.

In terms of extra work for the programmer, replication with voting, checkpointing and rollback, and algorithm-based fault tolerance require the algorithm to be redesigned in order to become fault tolerant. The self-stabilizing approach implies no extra burden for the programmer. The approach based on natural redundancy can be placed between these extremes. It requires some extra coding in order to add a recovery procedure to the algorithm, but no extensions to the algorithm are needed for the sake of creating redundant states.

The measurements of the time redundancy in fault-free situations for the three algorithms coded with the various techniques for fault tolerance are shown in Fig. 4. The experiments were carried out with three different sets of "basic" processors (4, 6 and 10 processors). By basic processors we mean those that are necessary for the execution of the normal (non-fault-tolerant) version of the algorithm. The algorithm-based version of the Markov algorithm that runs with 6 "basic" processors, needs in fact 7 processors because an additional processor is needed for calculating the checksum. The triplication with voting version of the Laplace algorithm that runs with 4 "basic" processors, needs in fact 12 processors. We also utilized different data sets in the experiments:

various grid sizes with the Laplace algorithm, various numbers of states with the Markov algorithm, and various numbers of variables with the linear equations algorithm. Also, the checkpointing and rollback versions of the algorithms were executed with different checkpointing intervals ($I = 50$, $I = 100$ and $I = 200$) which correspond to the number of supersteps in between two subsequent checkpoints (see Fig. 4).

In general the time overhead in fault-free situations decreases as the problem size increases. This is because, for larger data sets, the amount of computation due to the schemes for fault tolerance represent a smaller portion of the total computation of a superstep. The triplication with voting scheme was the one that required the highest amounts of time redundancy in fault-free situations, followed by the checkpointing and rollback technique. The algorithm-based fault tolerance technique and the approach based on natural redundancy incurred very small time overheads in fault-free situations. The advantage of the natural redundancy scheme is that it does not require any additional processors, besides requiring smaller time redundancy (in fault-free situations) than the algorithm-based approach.

The reasons why the replication with voting technique caused higher time overhead in fault-free situations are related to the increased data traffic in the bus and increased synchronization penalty implied by this technique. In a bus-based architecture machine, such as the Sequent Symmetry, the bus is a bottleneck. When one triplicates the number of processors utilized in a computation the data traffic is also triplicated, but still only one bus will have to handle that increased traffic. Furthermore, PPL synchronization primitives utilize busy wait. Processes needing to synchronize check constantly the value of a variable in shared memory. In order to access the value of this variable each process must use the bus. When one triplicates these accesses a price must be paid in terms of additional time overhead.

The time redundancy in fault-free situations for the checkpointing and rollback technique depends on how large is the size of the state that must be stored in each checkpoint. In our experiments this technique incurred larger time overheads with the implementation of the Laplace algorithm because the amount of data composing a checkpoint is larger for that algorithm than for the others. Another important parameter here is the checkpointing interval given by I, the number of supersteps in between checkpoints. Larger checkpointing intervals incur smaller values of time redundancy in fault-free situations, but require longer recovery times when faults do happen.

The natural redundancy version of the Laplace algorithm required less than 15% time redundancy in fault-free situations for problems of significant size. The Markov algorithm implementation incurred less than 8% time overhead in fault-free situations for all data sets utilized, while the linear equations algorithm implementation incurred less than 11% time overhead in the same conditions for all data sets utilized.

The self-stabilizing versions of the algorithms did not incur any performance overhead in the absence of faults. However, as discussed before, this technique can require very high time redundancy to recover from faults when they do occur.

Considering the tradeoffs between the various techniques for fault tolerance, the approach based on natural redundancy, provided this property is present in the application, results in the most attractive cost/benefit ratio if only single faults are likely to occur (which is true in most situations). It requires no extra processors, one superstep of time overhead for recovery, and low time overhead in fault-free situations with a small redesign effort of a given algorithm.

7 Conclusions

We have presented and discussed the results of practical experiments with three different algorithms implemented with several techniques for fault tolerance. The algorithms we utilized were: solution of Laplace equations by Jacobi's method, the computation of the invariant distribution of Markov chains, and the solution of systems of linear equations by Jacobi's method. The techniques for fault tolerance we used were: triplication with voting, checkpointing and rollback, self-stabilizing algorithms, algorithm-based fault tolerance, and natural redundancy. The implementations were realized on a bus-based shared memory architecture (Sequent Symmetry 12-processor computer) and were executed with several data sets and three different sets of processors.

The experimental results demonstrated that application-specific techniques are more likely to ensure fault-tolerant execution with the low amounts of space and time redundancy required by critical applications.

The fault detection schemes were shown to provide a fault coverage close to 100% for the simulated faults producing measurable errors in the final output.

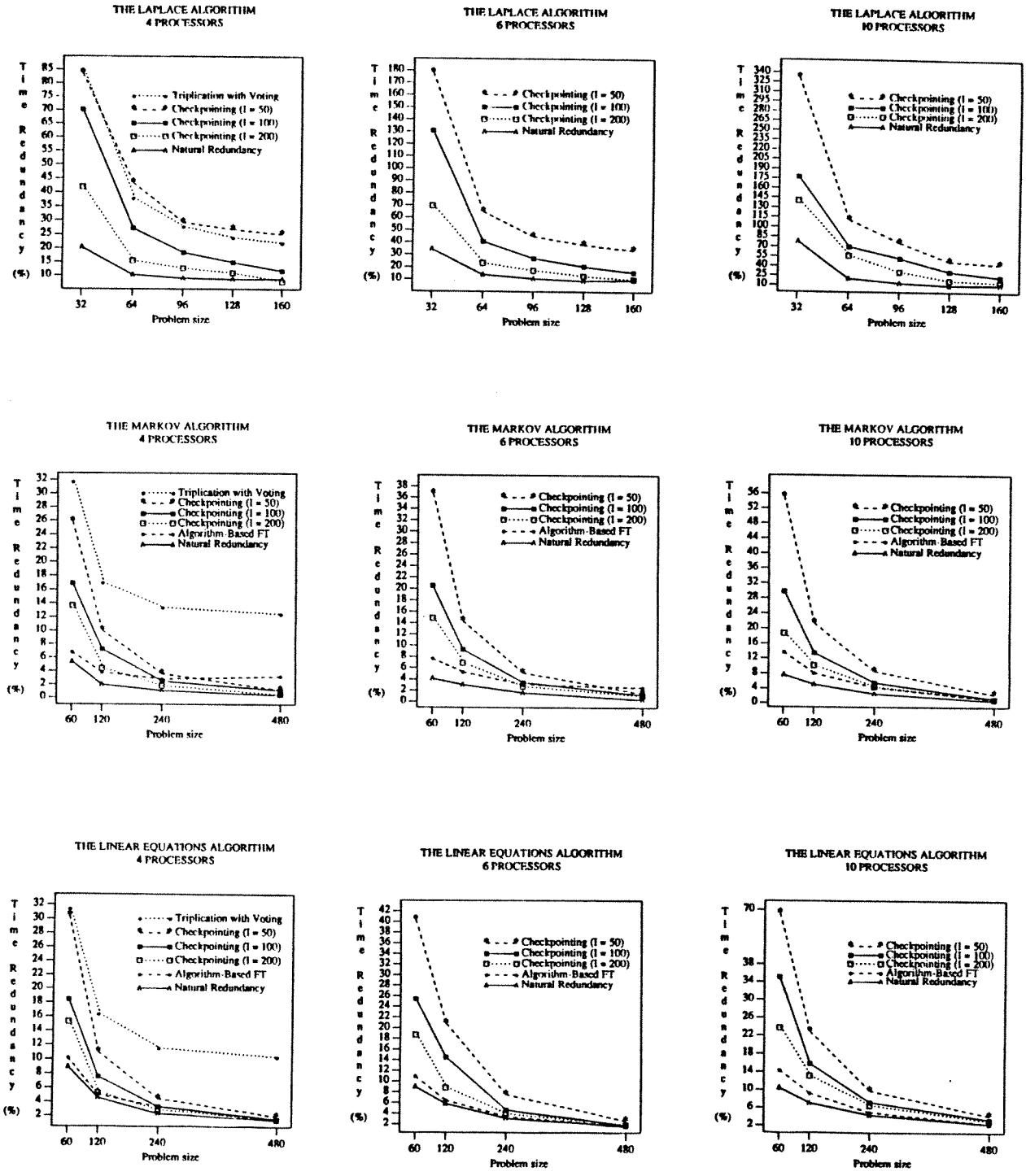


Figure 4: Comparison of time redundancy in fault-free situations for the three algorithms implemented with several techniques for fault tolerance running on four, six and ten "basic" processors.

More specifically, the approach based on natural redundancy proved to be especially attractive for critical applications implementation if only single faults are likely to occur. The implementation of the Laplace algorithm with this technique showed a time overhead in fault-free situations of less than 15% for significant size problems. The Markov algorithm implementation incurred less than 8% time overhead in fault-free situations for all data sets utilized, while the linear equations algorithm implementation incurred less than 10% time overhead in the same conditions for all data sets utilized except one (a 60-variables problem, executed with 10 processors, for which a 10.72% overhead was observed).

The results of our experiments confirm the philosophy which implies that fault-tolerant parallel/distributed systems can be successfully built by the development of an ultrareliable, formally-proved correct kernel and application-specific techniques for fault tolerance. This seems to be especially valid in the case of critical applications that must deliver continuous service in a timely manner.

References

[1] L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM,* vol. 33, no. 8, pp. 103-111, August 1990.

[2] F. Cristian, H. Aghili, R. Strong, D. Dolev, "Atomic Broadcast: From Simple Diffusion to Byzantine Agreement", *15th Int. Conference on Fault-Tolerant Computing*, pp. 200-206, 1985.

[3] D. P Siewiorek, R. S. Swarz, The Theory and Practice of Reliable System Design, Digital Press, 1982.

[4] R. Koo, S. Toueg, "Checkpointing and Rollback-Recovery for Distributed Systems," *IEEE Trans. on Software Engineering*, vol. SE-13, no. 1, pp. 23-31, January 1987.

[5] K. H. Huang, J. A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Software Engineering*, vol. SE-33, no. 6, pp. 518-528, June 1984.

[6] E. W. Dijkstra, "Self-Stabilizing Systems in Spite of Distributed Control," *Communications of the ACM*, vol. 17, no. 11, pp. 643-644, November 1974.

[7] L. Laranjeira, M. Malek, R. Jenevein, "On Tolerating Faults in Naturally Redundant Algorithms," *Proc. of 10th Symposium on Reliable Distributed Systems,* Pisa, Italy, pp. 118-127, September 1991.

[8] J. G. Kuhl, S. M. Reddy, "Fault Diagnosis in Fully Distributed Systems," *Proc. 11th Fault-Tolerant Computing Symposium*, pp. 100-105, June 1981.

[9] D. P. Bertsekas, J. N. Tsitsiklis, Parallel and Distributed Computation, Prentice-Hall, Englewood Cliffs, 1989.

[10] J. H. Patel, L. Y. Fung, "Concurrent Error Detection in ALUs by Recomputing with Shifted Operands," *IEEE Transactions on Computers*, vol. C-31, no. 7, pp. 589-595, July 1982.

[11] V. Balasubramanian, P. Banerjee, "Tradeoffs in the Design of Efficient Algorithm-Based Error Detection Schemes for Hypercube Multiprocessors," *IEEE Transactions on Software Engineering*, vol. SE-39, no. 2, February 1990.

[12] K. M. Chandy, "A Survey of Analytic Models of Rollback and Recovery Strategies," *IEEE Computer*, pp. 40-47, May 1975.

[13] K. M. Chandy, J. C. Browne, C. W. Dissly, W. R. Uhrig, "Analytic Models for Rollback and Recovery Strategies in Database Systems," *IEEE Transactions on Software Engineering*, Vol. 1, No. 1, pp. 100-110, March 1975.

TYPE OF TECHNIQUE	REDUNDANCY				FAULTS TOLERATED
	SPACE # of processors		TIME # of supersteps		
	needed	extra	needed	extra	
Triplication with Voting	$3N$	$2N$	$T+1$	1	multiple temporary and permanent
Checkpointing and Rollback	N	—	$T+I$	I	multiple temporary
Algorithm-based Fault Tolerance	$N+1$	1	$T+1$	1	single temporary
Self Stabilization	N	—	?	?	multiple temporary
Approach Based on Natural Redundancy	N	—	$T+1$	1	single temporary

Table 3: Necessary space redundancy, time redundancy needed for recovery, and types of faults tolerated by the techniques for fault tolerance used in the experiments.

SELF-STABILIZING LAPLACE ALGORITHM							
$NIC_{ff} = 6325$				$\epsilon = 0.1$			
bit_{er}	NIC_{fc}	bit_{er}	NIC_{fc}	bit_{er}	NIC_{fc}	bit_{er}	NIC_{fc}
0	6325	8	6325	16	6325	24	6399
1	6325	9	6325	17	6325	25	6370
2	6325	10	6325	18	6325	26	6977
3	6325	11	6325	19	6325	27	11202
4	6325	12	6325	20	6325	28	20691
5	6325	13	6325	21	6327	29	39674
6	6325	14	6325	22	6327	30	FE
7	6325	15	6325	23	6326	31	6289

Table 4: Number of supersteps necessary for the self-stabilizing (normal) Laplace algorithm to converge when different transient faults occur.

SELF-STABILIZING LINEAR EQUATIONS ALGORITHM							
$NIC_{ff} = 51$				$\epsilon = 0.0001$			
bit_{er}	NIC_{fc}	bit_{er}	NIC_{fc}	bit_{er}	NIC_{fc}	bit_{er}	NIC_{fc}
0	51	8	51	16	49	24	65
1	51	9	51	17	51	25	81
2	51	10	51	18	56	26	93
3	51	11	51	19	62	27	117
4	51	12	51	20	50	28	165
5	51	13	51	21	55	29	261
6	51	14	51	22	61	30	438
7	51	15	55	23	61	31	72

Table 5: Number of supersteps necessary for the self-stabilizing (normal) linear equations algorithm to converge when different transient faults occur.