Software-Implemented Fault Detection for High-Performance Space Applications

Michael Turmon, Robert Granat, and Daniel S. Katz
M/S 126-347; Jet Propulsion Laboratory; Pasadena, CA 91109
{turmon,granat,daniel.s.katz}@jpl.nasa.gov

Abstract

We describe and test a software approach to overcoming radiation-induced errors in spaceborne applications running on commercial off-the-shelf components. The approach uses checksum methods to validate results returned by a numerical subroutine operating subject to unpredictable errors in data. We can treat subroutines that return results satisfying a necessary condition having a linear form; the checksum tests compliance with this condition. We discuss the theory and practice of setting numerical tolerances to separate errors caused by a fault from those inherent in finite-precision numerical calculations. We test both the general effectiveness of the linear fault tolerant schemes we propose, and the correct behavior of our parallel implementation of them.

1 Introduction

We first outline the general outlook and goals of the spaceborne computing effort motivating this work, and then we describe the detailed contents of this paper.

1.1 Supercomputing in Space

Within NASA's High Performance Computing and Communications Program, the Remote Exploration and Experimentation (REE) project [11] at the Jet Propulsion Laboratory will enable a new type of scientific investigation by bringing commercial supercomputing technology into space. Transferring such computational power to space will enable highly-autonomous, flexible missions with substantial on-board analysis capability, mitigating control latency issues due to fundamental light-time delays, as well as inevitable bandwidth limitations in the link between spacecraft and ground stations. To do this, REE does not need to develop a single computational platform, but rather to define and demonstrate a process for rapidly transferring commercial high-performance computing technology into ultra-low power, fault-tolerant architectures for space.

Traditionally, spacecraft components have been radiation-hardened to protect against faults caused by natural galactic cosmic rays and energetic protons. Such radiation-hardening lowers the clock speed and increases the required power of a component. Even worse, the time needed to radiation-harden a component guarantees both that it will be outdated when it is ready for use in space, and that it has a high cost which must be spread over a small number of customers. Typically, at any given time, radiation-hardened components have a power:performance ratio that is an order of magnitude lower, and a cost that is several orders of magnitude higher than contemporary commodity off-the-shelf (COTS) components. The REE project is therefore attempting to use COTS components in space, and handling the resulting faults in software. The project consists of three initiatives: applications, computing testbeds, and system software.

Under the applications initiative, five Science Application Teams (SATs) were chosen to develop scalable science applications, and to port them to REE testbeds running REE system software:

- The Gamma-ray Large Area Space Telescope (GLAST) identifies gamma rays in a sea of background cosmic rays and reconstructs the gamma ray trajectories.
- The Next Generation Space Telescope (NGST) processes images on-board to reduce the effect of the cosmic rays on the CCD cameras. Also, fully on-board tuning of a deformable mirror allows accurate focus.
- Mars Rover Science, which identifies various materials on Mars using texture analysis and image segmentation. Also, stereo image pairs are analyzed for use in autonomous navigation.
- The Orbiting Thermal Imaging Spectrometer uses hyperspectral images to obtain temperature and emissivity, performs spectral unmixing, and classifies images.
- The Solar Terrestrial Probe Project (STP) examines using fleets of spacecraft for radio astronomical imaging and plasma moment analysis.

The power:performance and raw compute speed offered by

0-7695-0707-7/00 $10.00 © 2000 IEEE

REE allow these teams to develop new approaches to science data processing and autonomy. The applications mentioned above are generally MPI programs which are not replicated, and therefore can take full advantage of the computing power of the hardware. (REE also intends to support Triple Modular Redundancy in software for smaller applications that require high reliability, as opposed to high availability.)

Under the testbeds initiative, a system designed to deliver 30 MOPS/watt is currently being built, to be delivered in June 2000. This testbed consists of 40 COTS processors connected by a COTS network fabric. Through future RFPs, the project will obtain additional testbeds that perform faster while using less power. Criteria that are required of the testbeds are: consistency with rapid transfer (18 month or less) of new Earth-based technologies to space, no single point of failure, and graceful degradation in the event of permanent hardware failure.

The system software initiative will provide the services required to let the applications use the hardware reliably in space, as well as creating an easy-to-use development environment. The system software is also intended to use commercial components as much as possible. The major challenge for this initiative is to develop a middleware layer between the operating system and the applications which accepts that both permanent and transient faults will occur and provides for recovery from them.

1.2 Fault Tolerance via Software

Most of the transient faults will be single event upsets (SEUs); their presence requires that the applications be self-checking, or tolerant of errors, as the first layer of fault-tolerance. Additional software layers will protect against errors that are not caught by the application [5]. For example, one such layer would automatically restart programs which have crashed or hung. This works in conjunction with self-checking routines: if an error is detected, and the computation does not yield correct results after a set number of retries, the error handling scheme aborts the program so that it can be automatically restarted.

SEUs affecting data are particularly troublesome because they typically have fewer obvious consequences than an SEU to code — the latter would be expected to cause an exception. Note that since memory will be error-detecting and correcting, faults to memory will be largely screened; most data faults will therefore affect the microprocessor or its cache.

Due to the nature of scientific codes, much of their time is spent in certain common numerical subroutines — as much as 70% in one NGST application, for example. Protecting these subroutines from faults provides one ingredient in an overall software-implemented fault-tolerance scheme. It is in this context that we describe and test the mathematical background for using checksum methods to validate results returned by a numerical subroutine operating in an SEU-prone environment. Following the COTS philosophy laid out above, our general approach has been to wrap existing parallel numerical libraries (ScaLAPACK, FFTW) with fault-detecting middleware. We can treat subroutines that return results satisfying a necessary condition having a linear form; the checksum tests compliance with this necessary condition. Here we discuss the theory and practice of setting numerical tolerances to separate errors caused by a fault from those inherent in finite-precision numerical calculations.

To separate these two classes of errors, we employ well-known bounds on error-propagation within linear algebraic algorithms. These bounds provide a maximum error that is to be expected due to register effects; any error in excess of this is taken to be the product of a fault. Adapting these bounds to the fault tolerant software setting yields a series of tests having different efficiency and accuracy attributes. To better understand the characteristics of the tests we develop, we perform controlled numerical experiments using the tests, as well as experiments in an REE testbed environment, as described above, which supports software fault injection.

1.3 Notation

We close this introduction by introducing some useful notation. Matrices and vectors are written in uppercase and lowercase roman letters respectively; A^{T} is the transpose of the matrix A (conjugate transpose for complex matrices). Any identity matrix is always I; context provides its dimension. A is *orthogonal* if $AA^{\mathsf{T}} = I$. A square matrix is a *permutation* if it can be obtained by re-ordering the rows of I. The size of a vector v is measured by its *p-norm*, a non-negative real number $\|v\|_p$; similarly for matrices A. See [8] (hereafter abbreviated GVL), sections 2.2 and 2.3, for the definitions. The *submultiplicative property* of p-norms implies that $\|AB\|_p \leq \|A\|_p \|B\|_p$ and similarly for vectors.

2 General Considerations

In this paper we are concerned with these operations:

- Product: find the product $AB = P$, given A and B.
- LU decomposition: factor A as $A = PLU$ with P a permutation, L unit lower-triangular, U upper-triangular.
- Singular value decomposition: factor A as $A = UDV^{\mathsf{T}}$, where D is diagonal and U, V are orthogonal matrices.
- System solution: solve for x in $Ax = b$ when given A and b
- Matrix inverse: given A, find B such that $AB = I$.
- Fourier transform: given x, find y such that $y = Wx$, where W is the matrix of Fourier bases.
- Inverse Fourier transform: given y, find x such that $x = W^{\mathsf{T}}y$.

0-7695-0707-7/00 $10.00 © 2000 IEEE

Although standard numerical packages provide many other routines, the ones above were identified by science application teams as the being of the most interest, partly on the basis of amount of time spent within them.

Each of these operations has been written to emphasize that some linear relation holds among the subroutine inputs and its computed outputs; we call this the *postcondition*. For the product, system solution, inverse, and transforms, this postcondition is necessary and sufficient, and completely characterizes the subroutine's task. For the other two, the postcondition is only a necessary condition and valid results must enjoy other properties as well. In either case, identifying and checking the postcondition provides a powerful sanity check on the proper functioning of the subroutine.

Before proceeding to examine these operations in detail, we mention two points involved in designing fault tolerant techniques. Suppose for definiteness that we plan to check one $m \times n$ matrix. Any reasonable checksum scheme must depend on the content of each matrix entry, otherwise some entries would not be checked. This implies that simply computing a checksum requires $O(mn)$ operations. Checksum fault tolerance schemes thus lose their attractiveness for operations taking $O(mn)$ or fewer operations (e.g. trace, sum, and 1-norm) because it is simpler and more directly informative to achieve fault-tolerance by repeating the computation. The second general point is that, although the postconditions above are linearly-checkable equalities, they need not be. For example, the largest eigenvalue of A is bounded by functions of the 1-norm and the ∞-norm, both of which are easily computed but not linear. One could thus evaluate the sanity of a computation by checking postconditions that involve such inequalities. None of the operations we consider requires this level of generality.

The postconditions we consider generically involve comparing two linear maps, which are known in factorized form

$$L_1 L_2 \cdots L_p \overset{?}{=} R_1 R_2 \cdots R_q \quad . \tag{1}$$

This check can be done exhaustively via n linearly independent probes for an $n \times n$ system. Of course, exhaustive comparison would typically introduce about as much computation as would be required to recompute the answer from scratch. On the other hand, a typical fault to data fans out across the matrix outputs, and a single probe would be enough to catch most errors:

$$L_1 L_2 \cdots L_p w \overset{?}{=} R_1 R_2 \cdots R_q w \tag{2}$$

for some probe vector w. This approach, known as result-checking (RC), is recommended by Blum and Kannan [1] and, accessibly, Blum and Wasserman [3]. The idea is also the basis for the checksum-augmentation approach introduced earlier by Huang and Abraham [9] for systolic arrays, under the name algorithm-based fault tolerance (ABFT). Both techniques have since been extended and refined by several researchers [4, 7, 10, 13, 14, 2, 6]; a comparison of RC and ABFT is in [12].

There are two designer-selectable choices controlling the numerical properties of this fault detection system: the checksum weights w and the comparison method indicated above by $\overset{?}{=}$. When no assumptions may be made about the operands, the first is relatively straightforward: the elements of w should not vary greatly in magnitude so that results figure essentially equally in the check. At the minimum, w must be everywhere nonzero; better still, each partial product $L_{p'} \cdots L_p w$ and $R_{q'} \cdots L_q w$ of (2) should not vary greatly in magnitude. For Fourier transforms, this yields a weak condition on w and its transform — we have chosen a slowly decaying exponential which satisfies the condition. For the matrix operations, little can be said in advance about the factors so we are content to let w be the vector of all ones. Our implementation allows an arbitrary w to be supplied by those users with more knowledge of expected factors.

3 Error Propagation

After the checksum vector, the second choice is the comparison method. As stated above, we perform comparisons using the corresponding postcondition for each operation. To develop a test that is roughly independent of the matrices at hand, we use the well-known bounds on error propagation in linear operations. In what follows, we develop a test for each operation of interest. For each operation, we first cite a result bounding the numerical error in the computation's output, and then we use this bound to develop a corollary defining a test which is roughly independent of the operands. Those less interested in this machinery might review the first two results and skip to section 4. Throughout, we use $\mathbf{u}$ to represent the numerical precision of the underlying hardware; it is the difference between unity and the next larger floating-point number.

It is important to understand that the error bounds given in the results are *qualitative* and determine the general characteristics of roundoff in an algorithm's implementation. The estimates we obtain in this section are bounds based on worst-case scenarios, and will typically predict roundoff error larger than practically observed. (See GVL, section 2.4.6, for more on this outlook.) In the fault tolerance context, using these bounds uncritically would mean setting thresholds too high and missing some fault-induced errors. Their value for us, and it is substantial, is to indicate how roundoff error scales with different inputs. This allows fault tolerant routines the opportunity to factor out the inputs, yielding performance that is more nearly input-independent. Of course, some problem-specific tuning will likely improve performance. One goal is to simplify this tuning process as much as possible.

Result 1 *Let $\hat{P} = \texttt{mult}(A, B)$ be computed using a dot-product, outer-product, or gaxpy-based algorithm. The error matrix $E = \hat{P} - AB$ satisfies*

$$\|E\|_\infty \le n\|A\|_\infty\|B\|_\infty\mathbf{u} \tag{3}$$

Proof. See GVL, section 2.4.8. □

Corollary 2 *An input-independent checksum test for* `mult` *is*

$$d = \hat{P}w - ABw \tag{4}$$

$$\|d\|_\infty/(\|A\|_\infty\|B\|_\infty\|w\|_\infty) \gtrless \tau\mathbf{u} \tag{5}$$

where τ is an input-independent threshold.

The test is expressed as a comparison (indicated by the $\gtrless$ relation) with a threshold; the latter is a scaled version of the floating-point accuracy. If the discrepancy is larger than $\tau\mathbf{u}$, a fault would be declared, otherwise the error is explainable by roundoff.

Proof. The difference $d = Ew$ so, by the submultiplicative property of norms and result 1,

$$\|d\|_\infty \le \|E\|_\infty\|w\|_\infty \le n\|A\|_\infty\|B\|_\infty\|w\|_\infty\mathbf{u}$$

and the dependence on A and B is removed by dividing by their norms. The factor of n is unimportant in this calculation, as noted in the remark beginning the section. □

For the remaining operations, we require the notion of a *numerically realistic* matrix. The reliance of numerical analysts on certain proven algorithms is based on the rarity of certain pathological matrices that cause, for example, pivot elements in decomposition algorithms to grow exponentially. Even algorithms regarded as stable and reliable can be made to misbehave when given such unlikely inputs. Because the underlying routines will fail under such pathological conditions, we may neglect them in designing an fault tolerant system; such a computation is doomed even without faults. Accordingly, the results below must assume that the inputs are numerically realistic to obtain usable error bounds.

Result 3 *Let $(\hat{P}, \hat{L}, \hat{U}) = \texttt{lu}(A)$ be computed using a standard LU decomposition algorithm with partial pivoting. The backward error matrix E defined by $A + E = \hat{P}\hat{L}\hat{U}$ satisfies*

$$\|E\|_\infty \le 8n^3\rho\;\|A\|_\infty\mathbf{u} \tag{6}$$

where the growth factor *ρ depends on the size of certain partial results of the calculation, and is bounded by a small constant for numerically realistic matrices.*

Proof. See GVL, section 3.4.6. □

We note in passing that this is close to the best possible bound for the discrepancy, because the error in simply writing down the matrix A must be of order $\|A\|\mathbf{u}$.

Corollary 4 *An input-independent checksum test for* `lu` *as applied to numerically realistic matrices is*

$$d = \hat{P}\hat{L}\hat{U}w - Aw \tag{7}$$

$$\|d\|_\infty/(\|A\|_\infty\|w\|_\infty) \gtrless \tau\mathbf{u} \tag{8}$$

where τ is an input-independent threshold.

Proof. We have $d = Ew$ so, by the submultiplicative property of norms and result 3,

$$\|d\|_\infty \le \|E\|_\infty\|w\|_\infty \le 8n^3\rho\;\|A\|_\infty\|w\|_\infty\mathbf{u} \quad .$$

As before, the factor of $8n^3$ is unimportant in this calculation. For numerically realistic matrices, the growth factor ρ is bounded by a constant, and the indicated test is recovered by dividing by the norm of A. □

Result 5 *Let $(\hat{U}, \hat{D}, \hat{V}) = \texttt{svd}(A)$ be computed using a standard singular value decomposition algorithm. The forward error matrix E defined by $A + E = \hat{U}\hat{D}\hat{V}^\top$ satisfies*

$$\|E\|_2 \le \rho\;\|A\|_2\mathbf{u} \tag{9}$$

where ρ is a constant not much larger than one for numerically realistic matrices A.

Proof. See GVL, section 5.5.8. □

Corollary 6 *An input-independent checksum test for* `svd` *as applied to numerically realistic matrices is*

$$d = \hat{U}\hat{D}\hat{V}^\top w - Aw \tag{10}$$

$$\|d\|_\infty/(\|A\|_\infty\|w\|_\infty) \gtrless \tau\mathbf{u} \tag{11}$$

where τ is an input-independent threshold.

Proof. See the appendix. □

The test for SVD has the same normalization as for LU decomposition.

Result 7 *Let $\hat{B} = \texttt{inv}(A)$ be computed using Gaussian elimination with partial pivoting. The backward error matrix E defined by $(A + E)^{-1} = \hat{B}$ satisfies*

$$\|E\|_\infty \le 8n^3\rho\;\|A\|_\infty\mathbf{u} \tag{12}$$

with ρ as in result 3.

Proof. See GVL, section 3.4.6, which defines the backwards error for the linear system solution $Ax = b$. Since A^{-1} is calculated by solving the multiple right-hand-side problem $AA^{-1} = I$, the bound given there on $\|E\|_\infty$ applies here with the same growth factor ρ. (This growth factor depends only on the pivots in the LU factorization which underlies the inverse computation.) □

Algorithm	Δ	σ_1	σ_2	σ_3	Note
`mult`	$\hat{P} - AB$	$\|A\|\|B\|$	$\|\hat{P}\|$	$\|\hat{P}w\|$	—
`lu`	$\hat{P}\hat{L}\hat{U} - A$	$\|A\|$	$\|\hat{P}\hat{L}\hat{U}\|$	$\|Aw\|$	σ_1 easier than σ_2
`svd`	$\hat{U}\hat{D}\hat{V} - A$	$\|A\|$	$\|\hat{U}\hat{D}\hat{V}^{\mathsf{T}}\|$	$\|Aw\|$	σ_1 easier than σ_2
`inv`	$I - A\hat{B}$	$\|A\|\|A^{-1}\|$	$\|A\|\|\hat{B}\|$	$\|A\|\|\hat{B}w\|$	$\|A\hat{B}w\|$ useless
`fft`	$(\hat{y} - Wx)^{\mathsf{T}}$	$\|x\|$	—	—	result is a vector
`ifft`	$(\hat{x} - W^{\mathsf{T}}y)^{\mathsf{T}}$	$\|y\|$	—	—	result is a vector

Table 1. Algorithms and corresponding checksum tests.

Corollary 8 *An input-independent checksum test for* `inv` *as applied to numerically realistic matrices is*

$$d = w - A\hat{B}w \tag{13}$$

$$\|d\|_\infty / (\|A\|_\infty \|A^{-1}\|_\infty \|w\|_\infty) \gtrless \tau\mathbf{u} \tag{14}$$

where τ is an input-independent threshold.

Proof. See the appendix. □

We remark that this bound on discrepancy, larger than that for `lu`, is the reason matrix inverse is numerically unstable. We close this section with tests for Fourier transform operations. The $n \times n$ forward transform matrix W contains the Fourier basis functions, recall that $W/\sqrt{n}$ is unitary.

Result 9 *Let $\hat{y} =$ `fft`(x) be computed using a decimation-based fast Fourier transform algorithm; let $y = Wx$ be the infinite-precision Fourier transform. The error vector $e = \hat{y} - y$ satisfies*

$$\|e\|_\infty \leq n \log_2 n \, \|x\|_\infty \mathbf{u} \tag{15}$$

Proof. See the appendix. □

Corollary 10 *An input-independent checksum test for* `fft` *is*

$$d = (\hat{y} - Wx)^{\mathsf{T}} w \tag{16}$$

$$|d| / (\|x\|_\infty \|w\|_\infty) \gtrless \tau\mathbf{u} \tag{17}$$

where τ is an input-independent threshold.

Proof. This follows from result 9 after neglecting the leading constant. □

Corollary 11 *An input-independent checksum test for* `ifft` *is*

$$d = (\hat{x} - W^{\mathsf{T}}y)^{\mathsf{T}} w \tag{18}$$

$$|d| / (\|y\|_\infty \|w\|_\infty) \gtrless \tau\mathbf{u} \tag{19}$$

where τ is an input-independent threshold.

Proof. The proof, very similar to corollary 10, is omitted. □

4 Implementing the Tests

It is straightforward to transform these results into algorithms for error detection via checksums. The principal issue is computing the desired matrix norms efficiently from results needed in the root calculation. For example, in the matrix multiply, instead of computing $\|A\|\|B\|$, it is more efficient to compute $\|\hat{P}\|$ which equals $\|AB\|$ under fault-free conditions. By the submultiplicative property of norms, $\|AB\| \leq \|A\|\|B\|$, so this substitution always underestimates the upper bound on roundoff error, leading to false alarms. On the other hand, we must remember that the norm bounds are only general guides anyway. All that is needed is for $\|AB\|$ to scale as does $\|A\|\|B\|$; the unknown scale factor can be absorbed into τ.

Taking this one step farther, we might compute $\|\hat{P}w\|$ as a substitute for $\|A\|\|B\|\|w\|$. Here we run an even greater risk of underestimating the bound, especially if w is nearly orthogonal to the product, so it is wise to use instead $\lambda\|w\| + \|\hat{P}w\|$ for some problem-dependent λ. Extending this reasoning to the other operations yields the comparisons in table 1. The error criterion used there always proceeds from the number $\delta = \|\Delta w\|$ for the indicated difference matrix Δ; this matrix is of course never explicitly computed. In addition to the obvious

$$T0: \quad \delta / \|w\| \gtrless \tau\mathbf{u} \quad \text{(trivial test)} \tag{20}$$

we provide three other comparison tests

$$T1: \quad \delta / (\sigma_1 \|w\|) \gtrless \tau\mathbf{u} \quad \text{(ideal test)} \tag{21}$$

$$T2: \quad \delta / (\sigma_2 \|w\|) \gtrless \tau\mathbf{u} \quad \text{(approx. matrix test)} \tag{22}$$

$$T3: \quad \delta / (\lambda\|w\| + \sigma_3) \gtrless \tau\mathbf{u} \quad \text{(approx. vector test)} \tag{23}$$

The *ideal test* is the one recommended by the theoretical error bounds, and is based on the supplied input arguments, but may not be computable (e.g., for `inv`). In contrast, both approximate tests are based on computed quantities, and may also be suggested by the reasoning above. The *matrix test*

Average-case Matrices, All Faults

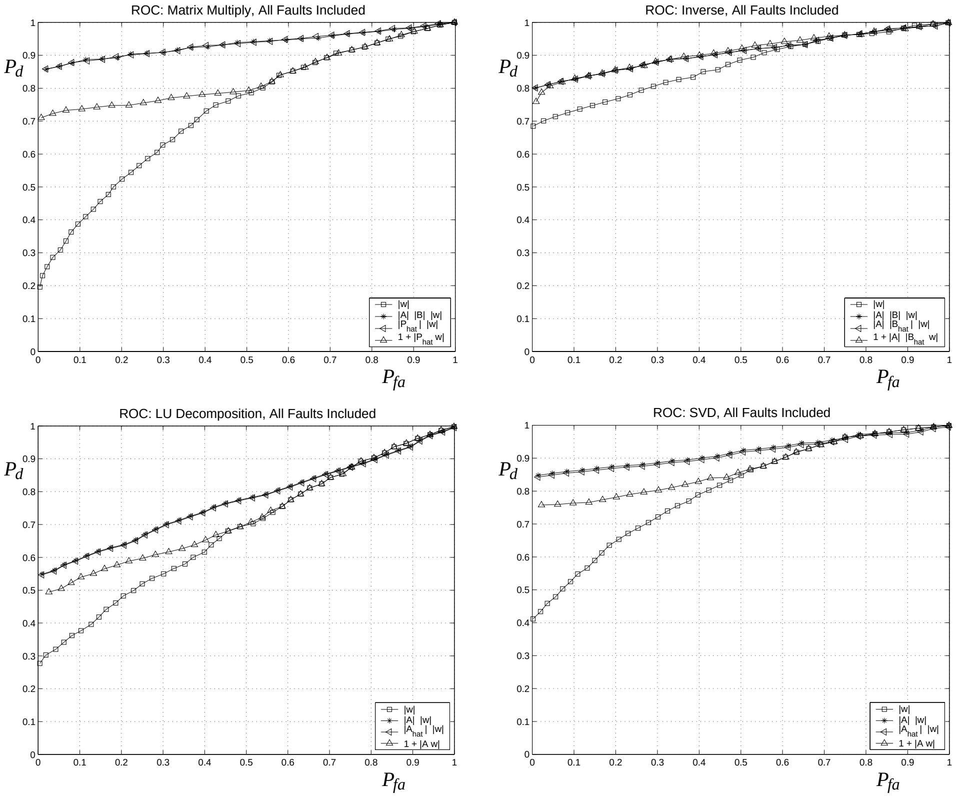

Figure 1. ROC for random matrices of bounded condition number, including all faults.

involves a matrix norm while the *vector test* involves a vector norm and is therefore more subject to false alarms. (Several variants of the matrix tests are available for these operations.) We note that the obvious vector test for `inv` uses $A\hat{B}w$, but since $\hat{B} = \texttt{inv}(A)$, this test becomes almost equivalent to $T0$: so we suggest using the vector/matrix test shown in table 1. The ideal tests $T1$ for the Fourier transforms need only the norm of the input, which is readily calculated, so other test versions are omitted. Clearly the choice of which test to use is based on the interplay of computation time and fault-detection performance for a given population of input matrices. Because of the shortcomings of numerical analysis, we cannot predict definitively that one test will outperform another. The experimental results reported in the next section are one indicator of real performance, and may motivate more detailed analysis of test behavior.

5 Results: Simulated Fault Conditions

We first discuss Matlab simulations of the checksum tests for the operations `mult`, `lu`, `svd`, and `inv` under fault conditions. We then discuss the behavior of a fault tolerant `fft` operation as implemented on a testbed environment, and tested using simulated fault injection.

5.1 Matlab Simulations

In this section we show results of Matlab simulations of the proposed checksum tests. These simulations are intended to verify the essential effectiveness of the checksum technique for fault tolerance, as well as to sketch the relative behaviors of the tests described above. Due to the special nature of the population of test matrices, and the shortcomings of the fault insertion scheme, these results should not be taken as anything but an estimate of relative performance, and a rough estimate of ultimate absolute performance.

We briefly describe the simulation setup. In essence a population of random matrices is used as input to a given computation; faults are injected in half these computations, and a checksum test is used to attempt to identify the affected computations. Random test matrices A of a given condition number κ are generated by the rule

$$A = 10^{\alpha}\, U D_{\kappa} V^{\mathsf{T}} \quad . \tag{24}$$

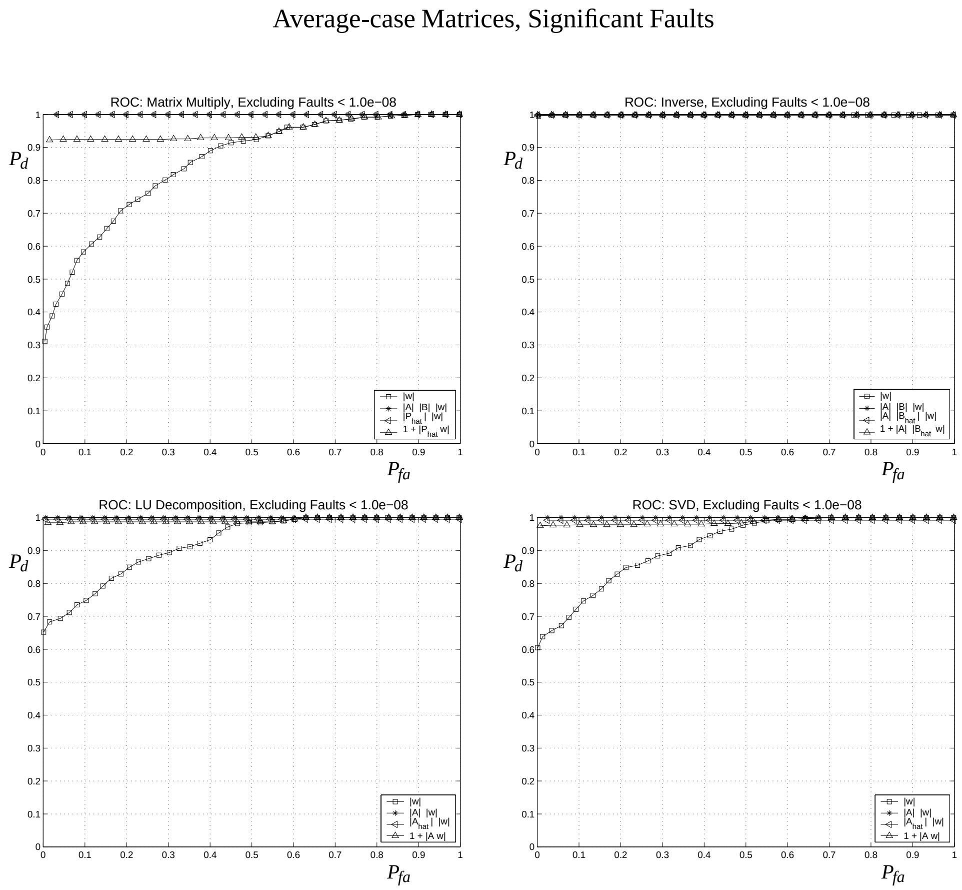

Figure 2. ROC for random matrices of bounded condition number, excluding faults of relative size less than 10^{-8}.

The random matrices U and V are the orthogonal factors in the QR factorization of two square matrices with normally distributed entries. The diagonal matrix D_κ is filled in by choosing random singular values, such that the largest singular value is unity and the smallest is $1/\kappa$. These matrices all have 2-norm equal to unity; the overall scale is set by α which is chosen uniformly at random between -8 and +8. A total of 2000 64 × 64 matrices (forty applications of the rule (24) for each κ in {2^1, ..., 2^{20}}) is processed.

Faults are injected in half of these runs (1000 of 2000) by first choosing a matrix to affect, and then flipping exactly one bit of its 64-bit representation. For example, if a call to If `lu` is to suffer a simulated fault, first one of A, L, or U is selected, and then one bit of the chosen matrix is toggled. If A was selected, one can expect the computed $\hat{L}$ and $\hat{U}$ to have many incorrect elements; if L was selected, only one element of the LU decomposition would be in error. This scheme is intended to simulate errors occurring at various times within the computation.

Characteristics of a given scheme are concisely expressed using the standard receiver operating characteristic (ROC) curve. For a given error tolerance, a certain proportion of False Alarms (numerical errors tagged as data faults, P_{fa}) and Detections (data faults correctly identified, P_d) will be observed. The ROC plots these two proportions parametrically as the tolerance is varied; this describes the performance achievable by a certain detection scheme and provides a basis for choosing one scheme over others.

Each of the four tests described above is used to identify faults; for a fixed τ this implies observing a certain false alarm rate and fault-detection rate. The pair (P_{fa}, P_d) may be plotted parametrically, varying τ to obtain an ROC curve which illustrates the overall performance of the test. See figure 1. In these figures, $T0$ is the line with square markers and $T3$ is marked by upward pointing triangles; $T0$ lies below $T3$. $T2$ is shown with left pointing triangles, and $T1$, the optimal test, with asterisks; these two tests nearly coincide.

Of course, some missed fault detections are worse than others since many faults occur in the low-order bits of the mantissa and cause very minor changes in the matrix. Accordingly, a second set of ROCs is shown in figure 2. In

0-7695-0707-7/00 $10.00 © 2000 IEEE

	Average-Case		Worst-Case	
	All	Sig.	All	Sig.
`mult`	0.85	1.00	0.63	0.92
`inv`	0.80	1.00	0.32	0.50
`lu`	0.54	1.00	0.43	0.90
`svd`	0.84	1.00	0.60	0.87
Mean	0.74	1.00	0.50	0.80

Table 2. P^* **for four sets of experiments.**

this set, faults which cause such a minute perturbation are screened from the results entirely; the screen is placed at a fault size of one part in 10^{-8} and filters about 40% of all faults. This corresponds to the accuracy of single-precision floating point and is well beyond the precision of the science data being analyzed. These ROCs are more informative about fault-detection performance in an operating regime similar to that of our science applications.

We may make some general observations about the results. Clearly $T0$, the un-normalized test, fares poorly in all experiments. This illustrates the value of the results on error propagation that form the basis for the normalized tests. Generally speaking,

$$T0 \ll T3 < T2 \approx T1 \quad . \tag{25}$$

This confirms theory, in which $T1$ is the ideal test and the others approximate it. In particular, $T1$ and $T2$ are quite similar because generally only an enormous fault can change the norm of a matrix — these cases are easy to detect.

Further, we note that the most relevant part of the ROC curve is when $P_{fa} \approx 0$; we may in fact be interested in the value P^*, defined to be P_d when $P_{fa} = 0$. This value is summarized for these experiments in table 2. The first two columns of this table come from the data in figures 1 and 2. The other columns are from similar experiments using a worst-case matrix population taken from the Matlab “gallery” matrices; this is a worst-case population because it contains many members that are not “numerically realistic” in the sense of section 3. Under the average-case test conditions, essentially all faults could be detected with no false alarms; this level of performance is surely adequate for REE purposes. In worst-case — and no science application should be in this regime — effectiveness drops to about 80%. This gives an indication of the loss in fault-detection performance incurred by a numerically ill-posed application.

5.2 Testbed Simulations

Effectiveness of the fault tolerant `fft` routine, implemented in C, was carried out on the REE interim testbed, a parallel system running the Lynx OS. Fault injection software available on the testbed was used to simulate radiation-induced SEUs affecting memory, registers, code, and the stack. A population of uniformly scaled random matrices was used as the input to the `fft` routine. The test was conducted in the following manner: each calculation was performed twice — once without the use of fault injection, and once while under the influence of simulated fault injection, during which zero or more faults were induced in memory as the calculation was being performed. The result with and without fault simulation was compared for each matrix, in order to verify whether a fault had been injected. For purposes of characterizing the performance of the fault detection, differences between the two results were considered insignificant (not faulty) if the square error was less than 10^{-8}. We note that this is similar, but not identical to the manner in which significant faults were identified in the preceding section.

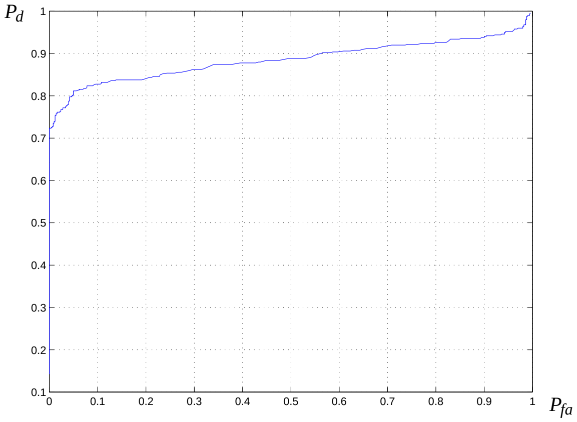

Figure 3. ROC for fault tolerant FFT.

Figure 3 shows an ROC curve summarizing the result of the tests for 1000 input matrices. This curve is not directly comparable to those in figures 1 and 2; nevertheless we observe that the fault tolerant `fft` does detect over 70% of errors without risk of false alarms. Our explanation of this result is inhibited by the limitations of the instrumentation: the fault injection software currently does not record the times, locations, or magnitudes of injected faults. However, one possible explanation for some faults going undetected is that the testbed allows faults to be injected in the matrix after it has been used in calculation of the checksum. Improvements in instrumentation should allow a more thorough analysis of these results. With increased understanding we believe these results can be improved.

6 Parallel Implementation

In our parallel implementation of the checksum procedures we use the ScaLAPACK routines `PDGEMM` for `mult`, `PDGETRF` for `lu`, `PDGETRI` for `inv`, and `PDGESVD` for `svd`. For `mult`, we use the checksum test $T2$ for reasons of computational cost, as the test requires only the calculation of the norm of the resultant matrix product. For lu and svd, we employ the ideal checksum test $T1$, as in these cases the norm of the matrix can be calculated before the factorization is performed in place. Our choice of checksum test for inv is complicated by the fact that PDGETRI requires that the input matrix already be in its LU-factorized form. We therefore employ a modified version of the checksum test $T3$, in which $\sigma_3 = \|\hat{B}\| \|Aw\|$: $\hat{B}$ is readily available as the result of the computation, and Aw can be obtained by multiplying w successively by $\hat{U}$, $\hat{L}$, and $\hat{P}$.

In our implementation we consider the possibility that induced faults could affect the calculated norms, thereby compromising the validity of the checksum test. To prevent erroneously large norms from eliminating errors from detection, the routines compare the norms against the system dependent maximum double precision floating point value; detection of a norm that exceeds that value raises an error.

In order to address numerical issues concerning our implementation of the checksum procedures, we compared the results of our implementation with those generated by Matlab computations. We performed a series of tests using an assortment of randomly perturbed matrices from the Matlab gallery selection. These matrices are generally ill-conditioned or poorly scaled, but serve as a demanding test to check our routines against a known standard.

From our tests, we ascertained that there is excellent agreement between Matlab and the ScaLAPACK implementation of our routines. Indeed, when the matrix is badly scaled, ill-conditioned, or numerically unrealistic — causing ScaLAPACK and Matlab to differ according to the full answer — our ScaLAPACK implementation finds the error in the checksum calculation also. In essence, the message is: if the computation did not succeed, the checksum test discovers it.

7 Conclusions and Future Work

Theoretical results bounding the expected roundoff error in a given computation provide several types of input-independent threshold tests for checksum differences. The observed behavior of these tests is in good general agreement with theory, and readily computable tests are easy to define. All the linear algebra operations considered here (mult, lu, inv, and svd) admit tests that are effective in detecting faults at the 99% level on typical matrix inputs. Tests of the numerical characteristics of our parallel implementation of the fault detection schemes indicate excellent agreement with another numerical package for most operations, except in cases when the matrix is badly scaled, ill-conditioned, or numerically unrealistic. In those cases, the schemes detect an error in the checksum calculation.

Test programs calling our parallel implementations have been installed on the REE project testbed, where they can be tested under simulated fault conditions. Of the operations described here, mult and fft have both been tested not only under the protection of the fault tolerant schemes described here, but also under an additional layer of software fault tolerance as described in Section 1.2.

The fault tolerant fft routines have also been integrated into the image texture analysis and segmentation application which is part of the Mars Rover Science project. This application is being tested with simulated fault injections on the REE project testbed under the software framework described above, both with and without the fault tolerant routines. While conclusive results are not yet available, preliminary testing indicates that the checksum scheme effectively protects the Fourier transform operations within the application from SEUs.

We expect that continued integration of fault tolerant routines with the various science applications will lead to these applications being resistant to SEUs throughout large portions of the computation. Other common subroutines, such as those involving sorting, order statistics, and numerical integration, also require more than $O(n)$ time and are candidates for fault-hardened versions. The fault-hardening described here is just one of the protections that will be needed to use COTS computers in space, but it is an essential one.

Acknowledgment

This work was carried out by the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

References

[1] M. Blum and S. Kannan. Designing programs that check their work. In *Proc. 21st Symp. Theor. Comput.*, pages 86–97, 1989.

[2] M. Blum, M. Luby, and R. Rubinfeld. Self-testing correcting with applications to numerical problems. *Journal of computer and system sciences*, 47(3):549–595, 1993.

[3] M. Blum and H. Wasserman. Reflections on the Pentium division bug. *IEEE Trans. Computing*, 45(4):385–393, 1996.

[4] D. L. Boley, R. P. Brent, G. H. Golub, and F. T. Luk. Algorithmic fault tolerance using the Lanczos method. *SIAM J. Matrix Anal. Appl.*, 13(1):312–332, 1992.

[5] F. Chen, L. Craymer, J. Deifik, A. J. Fogel, D. S. Katz, A. G. S. Jr., R. R. Some, S. A. Upchurch, and K. Whisnant. Demonstration of the REE fault-tolerant parallel-processing supercomputer for spacecraft onboard scientific data processing. In *Proc. ICDSN (FTCS-30 & DCCA-8)*, 2000.

[6] A.-R. Chowdhury and P. Banerjee. A new error analysis based method for tolerance computation for algorithm-based checks. *IEEE Trans. Computing*, 45(2), 1996.

[7] M. P. Connolly and P. Fitzpatrick. Fault-tolerent QRD recursive least squares. *IEE Proc. Comput. Digit. Tech.*, 143(2):137–144, 1996. (IEE, not IEEE).

[8] G. H. Golub and C. F. V. Loan. *Matrix Computations*. Johns Hopkins Univ., Baltimore, second edition, 1989.

[9] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Trans. Computing*, 33(6):518–528, 1984.

[10] F. T. Luk and H. Park. An analysis of algorithm-based fault tolerance techniques. *Journal of Parallel and Distributed Computing*, 5:172–184, 1988.

[11] REE project, March 1999. "Project Plan: Remote Exploration and Experimentation (REE) Project," available at `www-ree.jpl.nasa.gov`.

[12] P. Prata and J. G. Silva. Algorithm-based fault tolerance versus result-checking for matrix computations. In *Proc. FTCS-29*, pages 4–11, 1999.

[13] S. J. Wang and N. K. Jha. Algorithm-based fault tolerance for FFT networks. *IEEE Trans. Computing*, 43(7):849–854, 1994.

[14] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, 1997.

Appendix

Proof of Corollary 6. Let $d = \hat{U}\hat{D}\hat{V}^{\mathsf{T}} w - Aw$; then $d = Ew$ where E is the error matrix bounded in result 5. We claim that an input-independent checksum test for `svd` is

$$\|d\|_2/(\|A\|_2\|w\|_2) \gtrless \tau\mathbf{u} \quad . \tag{26}$$

Indeed, by the submultiplicative property and result 5,

$$\|d\|_2 \le \|E\|_2\|w\|_2 \le \rho\,\|A\|_2\|w\|_2\mathbf{u}$$

and the dependence on A is removed by dividing by its norm. The constant ρ is negligible for numerically realistic matrices, and the claim follows. To convert this test, which uses 2-norm, into one using the ∞-norm, we note that these two norms differ only by constant factors (see GVL sections 2.2.2 and 2.3.2) which may be absorbed into τ. □

Proof of Corollary 8. Note that $d = \Delta w$ where $\Delta = I - A(A+E)^{-1}$. Some algebra is necessary to extract the error E from Δ. Using the Sherman-Morrison formula (GVL section 2.1.3) to rewrite the inverse of $A + E$ we obtain

$$\begin{aligned}\Delta &= I - A[A^{-1} - A^{-1}(I + EA^{-1})^{-1}EA^{-1}]\\ &= (I + EA^{-1})^{-1}EA^{-1}\end{aligned} \tag{27}$$

For numerically realistic matrices, A dominates E and the first factor is negligible. Heuristically, this is because $E \ll A$ implies $EA^{-1} \ll AA^{-1} = I$, collapsing that factor to I. More formally, inverting a numerically realistic matrix produces an error matrix E such that for any vector v, $\|Ev\| \ll \|Av\|$ otherwise the backward error E would be comparable to A. Since v is arbitrary and A is invertible, we may let $v = A^{-1}u$, obtaining that $\|EA^{-1}u\| \ll \|u\| = \|Iu\|$, showing that the operator EA^{-1} is dominated by I. Therefore we may neglect the first factor and the norm of the error is bounded by

$$\begin{aligned}\|d\|_\infty &= \|\Delta w\|_\infty\\ &\le \|E\|_\infty\,\|A^{-1}\|_\infty\,\|w\|_\infty\\ &\le 8n^3\rho\,\|A\|_\infty\,\|A^{-1}\|_\infty\,\|w\|_\infty\mathbf{u}\end{aligned} \tag{28}$$

using the submultiplicative property of norms. As before, the factor of $8n^3$ is unimportant in this calculation. Invoking the assumption that A is a numerically realistic matrix allows us to neglect the growth factor ρ, yielding the indicated test. □

Proof of Result 9. Decimation algorithms are based on compact factorizations of the $n \times n$ unitary transform matrix W:

$$y = W_N W_{N-1} \cdots W_1\,x$$

where $N = \log_2 n$, and each W_k performs one bank of $n/2$ "butterfly" operations. The infinite-precision computation may therefore be written as a recurrence

$$\begin{aligned}z_0 &= x\\ z_{k+1} &= W_{k+1}\,z_k \quad (k \ge 0)\end{aligned} \tag{29}$$

where $y = z_N$. The finite-precision computation finds, in turn,

$$\begin{aligned}\hat{z}_0 &= x\\ \hat{z}_{k+1} &= \texttt{mult}(W_{k+1}, \hat{z}_k) \quad (k \ge 0)\end{aligned} \tag{30}$$

and $\hat{y} = \hat{z}_N$. The proof proceeds by developing a recurrence for the size (always expressed in ∞-norm) of the error vector

$$\begin{aligned}e_{k+1} &= z_{k+1} - \hat{z}_{k+1}\\ &= W_{k+1}\,z_k - \texttt{mult}(W_{k+1}, \hat{z}_k)\\ &= W_{k+1}\,z_k - (W_{k+1}\,\hat{z}_k + \tilde{e}_k)\\ &= W_{k+1}\,e_k - \tilde{e}_k\end{aligned} \tag{31}$$

where by Result 1, and the observation that exactly two entries of each row of W_k are nonzero, $\tilde{e}_k$ satisfies

$$\begin{aligned}\|\tilde{e}_k\| &\le 2\,\|W_{k+1}\|\,\|\hat{z}_k\|\mathbf{u}\\ &= 2\,\|z_k - e_k\|\mathbf{u}\\ &\le 2(\|z_k\| + \|e_k\|)\mathbf{u} \quad .\end{aligned} \tag{32}$$

Combining with (31) yields the bound

$$\begin{aligned}\|e_{k+1}\| &\le \|W_{k+1}\|\,\|e_k\| + 2(\|z_k\| + \|e_k\|)\mathbf{u}\\ &= (1 + 2\mathbf{u})\|e_k\| + 2\|z_k\|\mathbf{u} \quad .\end{aligned} \tag{33}$$

Since $\|W_k\| = 2$, $\|z_k\| \le 2^k\|x\|$, and we obtain the recurrent upper bound

$$\begin{aligned}\|e_0\| &= 0\\ \|e_{k+1}\| &\le (1 + 2\mathbf{u})\|e_k\| + 2^{k+1}\|x\|\mathbf{u} \quad (k{}^{3}0)\end{aligned} \tag{34}$$

For any reasonable floating-point system, $1+2\mathbf{u} \le 2$. Using this, it is easy to see $\|e_k\| \le k\,2^k\,\|x\|\mathbf{u}$, establishing the claim $\|e_N\| \le n\log n\,\|x\|\mathbf{u}$. □