Faster Numerical Algorithms via Exception Handling

James W. Demmel, *Member, IEEE*, and Xiaoye Li

***Abstract*—An attractive paradigm for building fast numerical algorithms is the following: 1) try a fast but occasionally unstable algorithm, 2) test the accuracy of the computed answer, and 3) recompute the answer slowly and accurately in the unlikely event it is necessary. This is especially attractive on parallel machines where the fastest algorithms may be less stable than the best serial algorithms. Since unstable algorithms can overflow or cause other exceptions, exception handling is needed to implement this paradigm safely. To implement it efficiently, exception handling cannot be too slow. We illustrate this paradigm with numerical linear algebra algorithms from the LAPACK library.**

***Index Terms*—IEEE floating point arithmetic, exception handling, linear algebra, LAPACK, speedup, NaN's, basic linear algebra subprograms.**

I. Introduction

A WIDELY accepted design paradigm for computer hardware is to execute the most common instructions as quickly as possible, and replace rarer instructions by sequences of more common ones. In this paper we explore the use of this paradigm in the design of numerical algorithms. We exploit the fact that there are numerical algorithms that run quickly and usually give the right answer as well as other, slower, algorithms that are always right. By "right answer" we mean that the algorithm is *stable,* or that it computes the exact answer for a problem that is a slight perturbation of its input [12]; this is all we can reasonably ask of most algorithms. To take advantage of the faster but occasionally unstable algorithms, we will use the following paradigm:

(1) use the fast algorithm to compute an answer; this will usually be done stably;
(2) quickly and reliably assess the accuracy of the computed answer;
(3) in the unlikely event the answer is not accurate enough, recompute it slowly but accurately.

The success of this approach depends on there being a large difference in speed between the fast and slow algorithms, on being able to measure the accuracy of the answer quickly and reliably, and, most important for us, on floating point exceptions not causing the unstable algorithm to abort or run very slowly. This last requirement means the system must either continue past exceptions and later permit the program to determine whether an exception occurred, or else support user-level trap handling. In this paper we will assume the first response to exceptions is available; this corresponds to the default behavior of IEEE standard floating point arithmetic [3], [4].

Our numerical methods will be drawn from the LAPACK library of numerical linear algebra routines for high performance computers [2]. In particular, we will consider condition estimation (error bounding) for linear systems, computing eigenvectors of general complex matrices, the symmetric tridiagonal eigenvalue problem, and the singular value decomposition. What the first two algorithms have in common is the need to solve triangular systems of linear equations which are possibly very ill-conditioned. Triangular system solving is one of the matrix operations found in the Basic Linear Algebra Subroutines, or BLAS [9], [10], [18]. The BLAS, which include related operations like dot product, matrix-vector multiplication, and matrix-matrix multiplication, occur frequently in scientific computing. This has led to their standardization and widespread implementation. In particular, most high performance machines have highly optimized implementations of the BLAS, and a good way to write portable high performance code is to express one's algorithm as a sequence of calls to the BLAS. This has been done systematically in LAPACK for most of numerical linear algebra, leading to significant speedups on highly pipelined and parallel machines [2].

However, the linear systems arising in condition estimation and eigenvector computation are often ill-conditioned, which means that over/underflow is not completely unlikely. Since the first distribution of LAPACK had to be portable to as many machines as possible, including those where all exceptions are fatal, it could not take advantage of the speed of the optimized BLAS, and instead used tests and scalings in inner loops to avoid computations that might cause exceptions.

In this paper, we present algorithms for condition estimation and eigenvector computation that use the optimized BLAS, test flags to detect when exceptions occur, and recover when exceptions occur. We report performance results on a "fast" DECstation 5000 and a "slow" DECstation 5000 (both have a MIPS R3000 chip as CPU [17]), a Sun 4/260 (which has a SPARC chip as CPU [15]), a DEC Alpha [11], a CRAY-C90 and a SPARCstation 10 with a Viking microprocessor. The "slow" DEC 5000 correctly implements IEEE arithmetic, but

Manuscript received October 14, 1993; revised march 1, 1994. J. W. Demmel was supported by NSF grant ASC-9005933, DARPA contract DAAL03-91-C-0047 via a subcontract from the University of Tennessee (administered by ARO), and DARPA grant DM28E04120 via a subcontract from Argonne National Laboratory. X. Li was supported by the National Science Foundation under award number ASC-9005933, and by Subcontract ORA4466.02 to the University of Tennessee (Defense Advanced Research Projects Administration contract number DAAL03-91-C-0047). This work is an expanded version of a paper published in the *Proc. 11th Symp. Comput. Arithmetic*, Windsor, ON Canada, June 29–July 2, 1993, ©1993, IEEE.

J. W. Demmel is with the Computer Science Division and Mathematics Department, University of California, Berkeley, CA 94720 USA; e-mail: demmel@cs.berkeley.edu.

X. Li is with the Computer Science Division, University of California, Berkeley, CA 94720 USA; e-mail: xiaoye@cs.berkeley.edu.

IEEE Log Number 9403194.

does arithmetic with NaN's about 80 times slower than normal arithmetic. The "fast" DEC 5000 implements IEEE arithmetic incorrectly, when the operands involve denormals or NaN's, but does so at the same speed as normal arithmetic. Otherwise, the two DEC 5000 workstations are equally fast[1]. The CRAY does not have exception handling, but we can still compare speeds in the most common case where no exceptions occur to see what speedup there could be if exception handling were available. We measure the *speedup* as the ratio of the time spent by the old LAPACK routine to the time spent by our new routine. The speedups we obtained for condition estimation in the most common case where no exceptions occur were as follows. The speedups ranged from 1.43 to 6.50 on either DEC 5000, from 1.50 to 5.00 on the Sun, from 1.66 to 9.28 on the DEC Alpha, and from 2.55 to 4.21 on the CRAY. Results for computing eigenvectors were about 1.08. These are quite attractive speedups. They would be even higher on a machine where the optimized BLAS had been parallelized but the slower scaling code had not.

In the rare case when exceptions did occur, the speed depended very strongly on whether the exception occurred early or late during the triangular solve, and on the speed of subsequent arithmetic with NaN (Not-a-Number) arguments. On some examples the speedup was as high as 5.41 on the fast DEC 5000, but up to 13 times *slower* on the slow DEC 5000. This illustrates the price of implementing IEEE NaN arithmetic too slowly.

We discuss the bisection algorithm for finding the eigenvalues of symmetric tridiagonal matrices. The LAPACK SSTEBZ routine takes special care in the inner loop to avoid overflow or division by zero, whereas our algorithm takes advantage of infinity arithmetic defined in the IEEE standard. We report performance results on a SPARCstation IPX (which has a Weitek 8601 chip as FPU), as well as on a distributed memory multiprocessor—the CM-5. The speedups range from 1.14 to 1.47.

We also discuss a singular value decomposition algorithm used in the LAPACK routine SBDSQR, where the careful scaling code can be avoided by using exception handling. The speedups we have obtained on a CRAY Y-MP (EL/2-256) were between 1.21 and 1.39.

The rest of this paper is organized as follows. Section II describes our model of exception handling in more detail. Section III describes the algorithms for solving triangular systems both with and without exception handling. Section IV describes the condition estimation algorithms both with and without exception handling, and gives timing results. Section V does the same for eigenvector computations. Section VI compares the bisection algorithms for solving the symmetric tridiagonal eigenvalue problem both with and without exception handling. Section VII describes the benefit from exception handling when computing singular values of a matrix. Section VIII draws lessons about the value of fast exception handling and fast arithmetic with NaN's and infinity symbols. Section IX suggests future research.

[1]Normally a buggy workstation would be annoying, but in this case it permitted us to run experiments where only the speed of exception handling varied.

TABLE I
THE IEEE STANDARD EXCEPTIONS AND THE DEFAULT VALUES

Exception raised	Default value	Condition
overflow	$\pm\infty$	$e > e_{max}$
underflow	$0, \pm 2^{e_{min}}$ or denormals	$e < e_{min}$
division by zero	$\pm\infty$	$x/0$, with finite $x \neq 0$
invalid	NaN	$\infty + (-\infty)$, $0 \times \infty$, $0/0$, ∞/∞, etc.
Inexact	round(x)	true result not representable

II. Exception Handling

In this section, we review how IEEE standard arithmetic handles exceptions, discuss how the relative speeds of its exception handling mechanisms affect algorithm design, and state the assumptions we have made about these speeds in this paper. We also briefly describe our exception handling interface on the DECstation 5000.

The IEEE standard classifies exceptions into five categories: *overflow*, *underflow*, *division by zero*, *invalid operation*, and *inexact*. Associated with each exception are a status flag and a trap. Each of the five exceptions will be signaled when detected. The signal entails setting a status flag, taking a trap, or possibly doing both. All the flags are "sticky," which means that after being raised they remain set until explicitly cleared. All flags can be tested, saved, restored, or altered explicitly by software. A trap should come under user control in the sense that the user should be able to specify a handler for it, although this capability is seldom implemented on current systems. The default response to these exceptions is to proceed without a trap and deliver to the destination an appropriate default value. The standard provides a clearly-defined default result for each possible exception. The default values and the conditions under which they are produced are summarized in Table I. Once produced, IEEE default behavior is for $\pm\infty$ and NaN to propagate through the computation without producing further exceptions.

According to the standard, the traps and sticky flags provide two different exception handling mechanisms. Their utility depends on how quickly and flexibly they permit exceptions to be handled. Since modern machines are heavily pipelined, it is typically very expensive or impossible to precisely interrupt an exceptional operation, branch to execute some other code, and later resume computation. Even without pipelining, operating system overhead may make trap handling very expensive. Even though no branching is strictly needed, merely testing sticky flags may be somewhat expensive, since pipelining may require a synchronization event in order to update them. Thus it appears fastest to use sticky flags instead of traps, and to test sticky flags as seldom as possible. On the other hand, infrequent testing of the sticky flags means possibly long stretches of arithmetic with $\pm\infty$ or NaN as arguments. If default IEEE arithmetic with them is too slow compared to arithmetic with normalized floating point numbers, then it is clearly inadvisable to wait too long between tests of the sticky flags to decide whether alternate computations should be performed. In summary, the fastest algorithm depends on

TABLE II
SPEED OF ARITHMETIC WITH DENORMAL, ∞ AND NaN AS ARGUMENTS, AS COMPARED WITH CONVENTIONAL ARITHMETIC

CPU	denormal	∞	NaN	how measured
MIPS R3000/3010				cc
"slow" (correct)	120x slower	full speed	80x slower	
"fast" (buggy)	"full speed"	full speed	"full speed" [2]	
MIPS R4000/4010	120x slower	full speed	32x slower	cc
Sun4/260 + Weitek 1164/5	8x slower	full speed	10x slower	f77
SPARC IPX + Weitek 8601	7x slower	full speed	9x slower	f77
SuperSPARC (Viking)	full speed	full speed	full speed	f77
PA-RISC	68x slower	full speed	42x slower	cc
RS/6000	full speed	full speed	full speed	cc
PowerPC	full speed	full speed	full speed	manual
387, 486, Pentium	full speed	full speed	full speed	manual
i860	868x slower	432x slower	411x slower	cc
i960	full speed	full speed	full speed	manual
DEC Alpha	690x slower	343x slower	457x slower	cc
CRAY-C90 (non IEEE machine)	N/A	abort	abort	manual

[2]Returns the first argument for binary operations; $\sqrt{-4} = 0$; status flag is not set.

the relative speeds of

conventional, unexceptional floating point arithmetic,

arithmetic with NaN's and $\pm\infty$ as arguments,

testing sticky flags, and

trap handling.

In the extreme case, where everything except conventional, unexceptional floating point arithmetic is terribly slow, we are forced to test and scale to avoid all exceptions. This is the unfortunate situation we were in before the introduction of exception handling, and it would be an unpleasant irony if exception handling were rendered too unattractive to use by too slow an implementation. In this paper, we will design our algorithms assuming that user-defined trap handlers are not available, that testing sticky flags is expensive enough that it should be done infrequently, and that arithmetic with NaN and $\pm\infty$ is reasonably fast. Our codes will in fact supply a way to measure the benefit one gets by making NaN and ∞ arithmetic fast. Table II shows the speed of arithmetic with denormalized numbers, ∞ and NaN, compared to conventional arithmetic on some machines. Some of the table entries are measured from Fortran, some from C, while others are from the architecture manuals. The DEC Alpha can only implement IEEE defaults, including infinities, NaN's and denormals, by precise interrupts; this causes significant loss of speed as compared with the normal arithmetic.

Our interface to the sticky flags is via subroutine calls, without special compiler support. We illustrate these interfaces briefly for one of our test machines, the DECstation 5000 with the MIPS R3000 chip as CPU. On the DECstation 5000, the R3010 Floating-Point Accelerator (FPA) operates as a coprocessor for the R3000 Processor chip, and extends the R3000's instruction set to perform floating point arithmetic operations. The FPA contains a 32-bit Control/Status register, FCR31, that is designed for exception handling and can be read/written by instructions running in User Mode. The FCR31 contains five *Nonsticky Exception* bits (one for each exception in Table I), which are appropriately set or cleared after every floating point operation. There are five corresponding *TrapEnable* bits used to enable a user level trap when an exception occurs. There are five corresponding *Sticky* bits to hold the accrued exception bits required by the IEEE standard for *trap disabled* operation. Unlike the nonsticky exception bits, the sticky bits are never cleared as a side-effect of any floating point operation; they can be cleared only by writing a new value into the Control/Status register. The nonsticky exception bits might be used in other applications requiring finer grained exception handling, such as parallel prefix [6].

In the algorithms developed in this paper we need only manipulate the trap enable bits (set them to zero to disable software traps) and the sticky bits. Procedure **exceptionreset()** clears the sticky flags associated with overflow, division by zero and invalid operations, and suppresses the exceptions accordingly. Function **except()** returns **true** if any or all of the overflow, division by zero and invalid sticky flags are raised.

III. TRIANGULAR SYSTEM SOLVING

We discuss two algorithms for solving triangular systems of equations. The first one is the simpler and faster of the two, and disregards the possibility of over/underflow. The second scales carefully to avoid over/underflow, and is the one currently used in LAPACK for condition estimation and eigenvector computation [1].

We will solve $Lx = b$, where L is a lower triangular n-by-n matrix. We use the notation $L(i : j, k : l)$ to indicate the submatrix of L lying in rows i through j and columns k through l of L. Similarly, $L(i, k : l)$ is the same as $L(i : i, k : l)$. The following algorithm accesses L by columns.

Algorithm 1: Solve a lower triangular system $Lx = b$.

```
$x(1:n) = b(1:n)$
for $i = 1$ to $n$
    $x(i) = x(i)/L(i,i)$
    $x(i+1:n) = x(i+1:n) - x(i) \cdot L(i+1:n,i)$
endfor
```

This is such a common operation that it has been standardized as subroutine STRSV, one of the BLAS [9], [10], [18]. Algorithm 1 can easily overflow even when the matrix L is well-scaled, i.e. all rows and columns are of equal and moderate length. For example,

$$x = L^{-1}b = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & c & 0 & 0 & 0 & 0 \\ 0 & -1 & c & 0 & 0 & 0 \\ 0 & 0 & -1 & c & 0 & 0 \\ 0 & 0 & 0 & -1 & c & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 \\ c^{-1} \\ c^{-2} \\ c^{-3} \\ c^{-4} \\ c^{-4} \end{bmatrix},$$

where $c = 10^{-10}$, overflows in IEEE single precision, even though each row and column of L has largest entry 1 in magnitude, and no terribly small entries. Similarly, let $L_n(c)$ be the analogous n-by-n matrix with $0 < c < 1$ in the second through $n-1$-st elements along the main diagonal. This means that $(L_n(c))^{-1}[1,0,\cdots,0]^T = [1,c^{-1},c^{-2},\cdots,c^{2-n},c^{2-n}]^T$.

The second algorithm scales carefully to avoid overflow in Algorithm 1. The algorithm works by choosing a scale factor $0 \le s \le 1$ and solving $Lx = sb$ instead of $Lx = b$. A value $s < 1$ is chosen whenever the solution x would overflow. In case x would overflow even if s were the smallest positive floating point number, s is set to zero (for example, consider $L_{27}(10^{-4})$ with IEEE single precision in the above example). If some $L(i,i) = 0$ exactly, so that L is singular, the algorithm will set $s = 0$ and compute a nonzero vector x satisfying $Lx = 0$ instead.

Here is a brief outline of the scaling algorithm; see [1] for details. Coarse bounds on the solution size are computed as follows. The algorithm begins by computing $c_j = \sum_{i=j+1}^{n} |L_{ij}|$, $G_0 = 1/\max_i |b_i|$, a lower bound G_i on the values of x_{i+1}^{-1} through x_n^{-1} after step i of Algorithm 1:

$$G_i = G_0 \prod_{j=1}^{i} \frac{|L_{jj}|}{|L_{jj}| + c_j},$$

and finally a lower bound g on the reciprocal of the largest intermediate or final values computed anywhere in Algorithm 1:

$$g = \min_{1 \le i \le n} (G_0, G_{i-1} \cdot \min(1, |L_{ii}|)) \quad .$$

Lower bounds on x_j^{-1} are computed instead of upper bounds on x_j to avoid the possibility of overflow in the upper bounds.

Let UN = 1/OV be smallest floating point number that can safely be inverted. If $g \ge$ UN, this means the solution can be computed without danger of overflow, so we can simply call the BLAS. Otherwise, the algorithm makes a complicated series of tests and scalings as in Algorithm 2.

Now we compare the costs of Algorithms 1 and 2. Algorithm 1 costs about n^2 flops (floating point operations), half additions and half multiplies. There are also n divisions which are insignificant for large n. In the first step of Algorithm 2, computing the c_i costs $n^2/2 + O(n)$ flops, half as much as Algorithm 1. In some of our applications, we expect to solve several systems with the same coefficient matrix, and so can reuse the c_i; this amortizes the cost over several calls. In the best case, when $g \ge$ UN, we then simply call STRSV. This makes the overall operation count about $1.5n^2$ (or n^2 if we amortize). In the worst (and very rare) case, the inner loop of Algorithm 2 will scale at each step, increasing the operation count by about n^2 again, for a total of $2.5n^2$ (or $2n^2$ if we amortize). Updating $x_{\max}$ costs another $n^2/2$ data accesses and comparisons, which may or may not be cheaper than the same number of floating point operations.

More important than these operation counts is that Algorithm 2 has many data dependent branches, which makes it harder to optimize on pipelined or parallel architectures than the much simpler Algorithm 1. This will be borne out by the results in later sections.

Algorithm 2 is available as **LAPACK** subroutine SLATRS. This code handles upper and lower triangular matrices, permits solving with the input matrix or its transpose, and handles either general or unit triangular matrices. It is 300 lines long excluding comments. The Fortran implementation of the BLAS routine STRSV, which handles the same input options, is 159 lines long, excluding comments. For more details on SLATRS, see [1].

Algorithm 2: Solve a lower triangular system $Lx = sb$ with scale factor $0 \le s \le 1$.

```
Compute $g$ and $c_1,\cdots,c_{n-1}$ as described above
if ($g \ge$ UN) then
    call the BLAS routine STRSV
else
    $s = 1$
    $x(1:n) = b(1:n)$
    $x_{\max} = \max_{1\le i\le n} |x(i)|$
    for $i = 1$ to $n$
        if (UN $\le |L(i,i)| < 1$ and $|x(i)| > |L(i,i)| \cdot$ OV)
        then
            $scale = 1/|x(i)|$
            $s = s \cdot scale$; $x(1:n) = x(1:n) \cdot scale$;
            $x_{\max} = x_{\max} \cdot scale$
        else if ($0 > |L(i,i)| >$ UN and $|x(i)| > |L(i,i)| \cdot$
        OV) then
            $scale = ((|L(i,i)| \cdot \text{OV})/|x(i)|)/\max(1,c_i)$
            $s = s \cdot scale$; $x(1:n) = x(1:n) \cdot scale$;
            $x_{\max} = x_{\max} \cdot scale$
        else if ($L(i,i) = 0$) then ... compute a null vector
        $x$: $Lx = 0$
            $s = 0$
            $x(1:n) = 0$; $x(i) = 1$; $x_{\max} = 0$
        end if
        $x(i) = x(i)/L(i,i)$
        if ($|x(i)| > 1$ and $c(i) > (\text{OV} - x_{\max})/|x(i)|$) then
            $scale = 1/(2 \cdot |x(i)|)$
            $s = s \cdot scale$; $x(1:n) = x(1:n) \cdot scale$
        else if ($|x(i)| \le 1$ and $|x(i)| \cdot c(i) > (\text{OV} - x_{\max})$)
        then
            $scale = 1/2$
            $s = s \cdot scale$; $x(1:n) = x(1:n) \cdot scale$
        endif
        $x(i+1:n) = x(i+1:n) - x(i) \cdot L(i+1:n,i)$
        $x_{\max} = \max_{i<j\le n} |x(j)|$
    endfor
endif
```

IV. Condition Estimation

In this section, we discuss how IEEE exception handling can be used to design a faster condition estimation algorithm. We compare first theoretically and then in practice the old algorithm used in LAPACK with our new algorithm.

A. Algorithms

When solving the n-by-n linear system $Ax = b$, we wish to compute a bound on the error $x_{\text{computed}} - x_{\text{true}}$. We will measure the error using either the one-norm $\|x\|_1 = \sum_{i=1}^n |x_i|$, or the infinity norm $\|x\|_\infty = \max_i |x_i|$. Then the usual error bound [12] is

$$\|x_{\text{computed}} - x_{\text{true}}\|_1 \le k_1(A) \cdot p(n) \cdot \epsilon \cdot \rho \cdot \|x_{\text{true}}\|_1, \quad (1)$$

where $p(n)$ is a slowly growing function of n (usually about n), ϵ is the machine precision, $k_1(A)$ is the *condition number* of A, and ρ is the *pivot growth factor*. The condition number is defined as $k_1(A) = \|A\|_1 \cdot \|A^{-1}\|_1$, where $\|B\|_1 \equiv \max_{1\le j\le n} \sum_{i=1}^n |b_{ij}|$. Since computing A^{-1} costs more than solving $Ax = b$, we prefer to estimate $\|A^{-1}\|_1$ inexpensively from A's LU factorization; this is called *condition estimation*. Since $\|A\|_1$ is easy to compute, we focus on estimating $\|A^{-1}\|_1$. The pivot growth may be defined as $\frac{\|U\|_1}{\|A\|_1}$ (other definitions are possible). This is close to unity except for pathological cases.

In the LAPACK library [2], a set of routines have been developed to estimate the reciprocal of the condition number $k_1(A)$. We estimate the reciprocal of $k_1(A)$, which we call RCOND, to avoid overflow in $k_1(A)$. The inputs to these routines include the factors L and U from the factorization $A = LU$ and $\|A\|_1$. Higham's modification [14] of Hager's method [13] is used to estimate $\|A^{-1}\|_1$. The algorithm is derived from a convex optimization approach, and is based on the observation that the maximal value of the function $f(x) = \|Bx\|_1/\|x\|_1$ equals $\|B\|_1$ and is attained at one of the vectors e_j, for $j = 1, \cdots, n$, where e_j is the jth column of the n-by-n identity matrix.

Algorithm 3 [13]: This algorithm computes a lower bound γ for $\|A^{-1}\|_1$.

```
Choose $x$ with $\|x\|_1 = 1$ (e.g., $x := \frac{(1,1,\cdots,1)^T}{n}$)
Repeat
    solve $Ay = x$ (by solving $Lw = x$ and $Uy = w$
    using Algorithm 2)
    form $\xi :=$ sign($y$)
    solve $A^T z = \xi$ (by solving $U^T w = \xi$ and $L^T z = w$
    using Algorithm 2)
    if $\|z\|_\infty \le z^T x$ then
        $\gamma := \|y\|_1$
        quit
    else $x := e_j$, for that $j$ where $|z_j| = \|z\|_\infty$
```

The algorithm involves repeatedly solving upper or lower triangular systems until a certain stopping criterion is met. Due to the possibilities of overflow, division by zero, and invalid exceptions caused by the ill-conditioning or bad scaling of the linear systems, the LAPACK routine SGECON uses Algorithm 2 instead of Algorithm 1 to solve triangular systems like $Lw = x$, as discussed in Section 3. The details of the use of the scale factor s returned by Algorithm 2 are not shown; see routines SGECON and SLACON in LAPACK [2].

Our goal is to avoid the slower Algorithm 2 by using exception handling to deal with these ill-conditioned or badly scaled matrices. Our algorithm only calls the BLAS routine STRSV, and has the property that overflow occurs only if the matrix is extremely ill-conditioned. In this case, which we detect using the sticky exception flags, we can immediately terminate with a well-deserved estimate RCOND = 0. Merely replacing the triangular solver used in Algorithm 3 by STRSV and inserting tests for overflow does not work, as can be seen by choosing a moderately ill-conditioned matrix of norm near the underflow threshold; this will cause overflow while solving $Uy = w$ even though A is only moderately ill-conditioned. Therefore, we have modified the logic of the algorithm as follows. Comments indicate the guaranteed lower bound on $k_1(A)$ if an exception leads to early termination.

Algorithm 4: This algorithm estimates the reciprocal of $k_1(A) = \|A\|_1\|A^{-1}\|_1$.

```
Let    $\alpha = \|A\|_1$
RCOND is the estimated reciprocal of condition
number $k_1(A)$
Call **exceptionreset()**
Choose $x$ with $\|x\|_1 = 1$ (e.g., $x := \frac{(1,1,\cdots,1)^T}{n}$)
Repeat
    solve $Lw = x$ by calling STRSV
    if (**except()**) then RCOND := 0; quit /* $k_1(A) \ge$
    OV/$\rho$ */
    if ($\alpha > 1$) then
        if ($\|w\|_\infty \ge$ OV/$\alpha$) then
            solve $Uy = w$ by calling STRSV
            if (**except()**) then RCOND := 0; quit /*
            $k_1(A) \ge$ OV */
            else $y := y \cdot \alpha$
                if (**except()**) then RCOND := 0;
                quit /*$k_1(A) \ge$ OV */
            endif
        else solve $Uy = w \cdot \alpha$ by calling STRSV
            if (**except()**) then RCOND := 0; quit /*
            $k_1(A) \ge$ OV */
        endif
    else solve $Uy = w \cdot \alpha$ by calling STRSV
        if (**except()**) then RCOND := 0; quit /*
        $k_1(A) \ge$ OV */
    endif
    form $\xi :=$ sign($y$)
    solve $U^T w = \xi \cdot \alpha$ by calling STRSV
    if (**except()**) then RCOND := 0; quit /*
    $k_1(A) \ge \frac{\text{OV}}{n^3}$ */
    else solve $L^T z = w$ by calling STRSV
        if (**except()**) then RCOND:=0; quit /* $k_1(A)$
        $\ge \frac{\text{OV}}{n^2}$ */
    endif
    if $\|z\|_\infty \le z^T x$ then
        RCOND := $1/\|y\|_1$; quit
    else $x := e_j$, where $|z_j| = \|z\|_\infty$
    endif
```

The behavior of Algorithm 4 is described by the following:

Lemma 1: If Algorithm 4 stops early because of an exception, then the "true rounded" reciprocal of the condition number satisfies $RCOND \le \frac{\max(n^3,\rho)}{OV}$, where $\rho = \frac{\|U\|_1}{\|A\|_1}$ is the pivot growth factor.

Proof: In the algorithm, there are seven places where exceptions may occur. We will analyze them one by one. Note that x is chosen such that $\|x\|_1 = 1$, and that $\|\xi\|_1 = n$.

1) An exception occurs when computing $L^{-1}x$.
Since $A = LU$, $L^{-1} = UA^{-1}$, this implies

$$\begin{aligned}\mathrm{OV} &\le \|L^{-1}x\|_1 \le \|U\|_1\|A^{-1}\|_1\|x\|_1\\ &= \frac{\|U\|_1}{\|A\|_1}\|A\|_1\|A^{-1}\|_1 = \rho\cdot k_1(A).\end{aligned}$$

Therefore, $k_1(A) \ge \mathrm{OV}/\rho$, i.e., RCOND $\le \rho/\mathrm{OV}$.

2) An exception occurs when computing $U^{-1}L^{-1}x$ with $\alpha > 1$. Then

$$\mathrm{OV} \le \|U^{-1}L^{-1}x\|_1 \le \|A^{-1}\|_1 < \|A^{-1}\|_1\alpha = k_1(A),$$

so RCOND $\le 1/\mathrm{OV}$.

3) An exception occurs when computing $\alpha\cdot U^{-1}L^{-1}x$ with $\alpha > 1$. Then

$$\mathrm{OV} \le \|\alpha U^{-1}L^{-1}x\|_1 \le k_1(A),$$

so RCOND $\le 1/\mathrm{OV}$.

4) An exception occurs when computing $U^{-1}\alpha L^{-1}x$, with $\alpha > 1$ and $\|L^{-1}x\|_1 < \mathrm{OV}/\alpha$. Then

$$\mathrm{OV} \le \|U^{-1}\alpha L^{-1}x\|_1 \le \|A^{-1}\|_1\alpha = k_1(A),$$

so $k_1(A) \ge \mathrm{OV}$, i.e., RCOND $\le 1/\mathrm{OV}$.

5) An exception occurs when computing $U^{-1}\alpha L^{-1}x$ with $\alpha \le 1$. Then

$$\mathrm{OV} \le \|U^{-1}\alpha L^{-1}x\|_1 \le \|A^{-1}\|_1\alpha = k_1(A),$$

so $k_1(A) \ge \mathrm{OV}$, i.e., RCOND $\le 1/\mathrm{OV}$.

6) An exception occurs when computing $U^{-T}\alpha\xi$.
Since $A^T = U^TL^T$, $U^{-T} = L^TA^{-T}$, and $\|B^T\|_1 \le n\|B\|_1$, we get

$$\begin{aligned}\mathrm{OV} &\le \|U^{-T}\alpha\xi\|_1 \le \|L^T\|_1\|A^{-T}\|_1\alpha\|\xi\|_1\\ &\le \|L^T\|_1\cdot n\|A^{-1}\|_1\cdot\alpha\cdot\|\xi\|_1 = \|L^T\|_1\cdot n\cdot k_1(A)\cdot n\\ &\le n^3k_1(A).\end{aligned}$$

Therefore, $k_1(A) \ge \frac{\mathrm{OV}}{n^3}$, i.e., RCOND $\le \frac{n^3}{\mathrm{OV}}$.

7) An exception occurs when computing $L^{-T}U^{-T}\alpha\xi$, so

$$\begin{aligned}\mathrm{OV} &\le \|L^{-T}U^{-T}\alpha\xi\|_1 \le \|A^{-T}\|_1\alpha\|\xi\|_1\\ &\le n\|A^{-1}\|_1\cdot\alpha\cdot n = n^2k_1(A).\end{aligned}$$

Therefore, RCOND $\le \frac{n^2}{\mathrm{OV}}$.

Combining the above seven cases, we have shown that RCOND $\le \frac{\max(n^3,\rho)}{\mathrm{OV}}$ when an exception occurs. □

In practice, any RCOND $< \epsilon$ signals a system so ill-conditioned as to make the error bound in (1) as large as the solution itself or larger; this means the computed solution has no digits guaranteed correct. Since $\frac{\max(n^3,\rho)}{\mathrm{OV}} \ll \epsilon$ unless either n or ρ is enormous (both of which also mean the error bound in (1) is enormous), there is no loss of information in stopping early with RCOND $= 0$.

Algorithm 4 and Lemma 1 are applicable to any linear systems for which we do partial or complete pivoting during Gaussian elimination, for example, LAPACK routines SGECON, SGBCON and STRCON (see Section 4.2 for the descriptions of these routines), and their complex counterparts.

For symmetric positive definite matrices, where no pivoting is necessary, an analogous algorithm (e.g., SPOCON) was developed and analyzed, though omitted in this paper due to the limitation of the length.

B. Numerical Results

To compare the efficiencies of Algorithms 3 and 4, we rewrote several condition estimation routines in LAPACK using Algorithm 4, including SGECON for general dense matrices, SPOCON for dense symmetric positive definite matrices, SGBCON for general band matrices, and STRCON for triangular matrices, all in IEEE single precision. To compare the speed and the robustness of Algorithms 3 and 4, we generated various input matrices yielding unexceptional executions with or without invocation of the scalings inside Algorithm 2, as well as exceptional executions. The unexceptional inputs tell us the speedup in the most common case, and on machines like the CRAY measure the performance lost for lack of any exception handling.

First, we ran Algorithms 3 and 4 on a suite of well-conditioned random matrices where no exceptions occur, and no scaling is necessary in Algorithm 2. This is by far the most common case in practice. The experiments were carried out on a DECstation 5000, a SUN 4/260, a DEC Alpha, and a single processor CRAY-C90. The performance results are presented in Table III. The numbers in the table are the ratios of the time spent by the old LAPACK routines using Algorithm 3 to the time spent by the new routines using Algorithm 4. These ratios measure the *speedups* attained via exception handling. The estimated condition numbers output by the two algorithms are always the same. For dense matrices or matrices with large bandwidth, as matrix dimension n increases, the time to service cache misses constitutes a larger portion of the execution time, resulting in decreased speedups. When we ran SGBCON with matrices of small bandwidth, such that the whole matrix fit in the cache, we observed even better speedups.

Second, we compared Algorithms 3 and 4 on several intentionally ill-scaled linear systems for which some of the scalings inside Algorithm 2 have to be invoked, but whose condition numbers are still finite. For SGECON alone with matrices of sizes 100 to 500, we obtained speedups from 1.62 to 3.33 on the DECstation 5000, and from 1.89 to 2.67 on the DEC Alpha.

Third, to study the behavior and performance of the two algorithms when exceptions do occur, we generated a suite of ill-conditioned matrices that cause all possible exceptional paths in Algorithm 4 to be executed. Both Algorithms 3 and 4 consistently deliver zero as the reciprocal condition number. For Algorithm 4, inside the triangular solve, the computation involves such numbers as NaN and $\pm\infty$. Indeed, after an overflow produces $\pm\infty$, the most common situation is to subtract two infinities shortly thereafter, resulting in a NaN which then propagates through all succeeding operations. In other words, if there is one exceptional operation, the most common situation is to have a long succession of operations with NaN's. We compared the performance of the "fast" and

TABLE III
SPEEDUPS ON DEC 5000/SUN 4-260/DECALPHA/CRAY-C90

Machine	Matrix size n	100	200	300	400	500
DEC 5000	SGBCON $sbw = 4$	3.00	4.25	5.33	6.50	6.45
	$sbw = 0.8n$	1.57	1.46	1.55	1.56	1.67
	SGECON	2.00	1.52	1.46	1.44	1.43
	SPOCON	2.83	1.92	1.71	1.55	1.52
	STRCON	3.33	1.78	1.60	1.54	1.52
Sun 4/260	SGBCON	2.00	2.20	2.11	2.77	2.71
	SGECON	3.02	2.14	1.88	1.63	1.62
	SPOCON	5.00	2.56	2.27	2.22	2.17
	STRCON	1.50	2.00	2.30	2.17	2.18
DEC Alpha	SGBCON $sbw = 3$	2.00	2.00	8.67	8.40	9.28
	$sbw = 0.8n$	2.67	2.63	2.78	2.89	3.23
	SGECON	2.66	2.01	1.85	1.78	1.66
	SPOCON	2.25	2.46	2.52	2.42	2.35
	STRCON	3.00	2.33	2.28	2.18	2.07
CRAY-C90	SGECON	4.21	3.48	3.05	2.76	2.55

No exceptions nor scaling occur. SBW stands for semi-bandwidth.

TABLE IV
THE SPEEDS OF SOME EXAMPLES WITH EXCEPTIONS

	Example 1	Example 2	Example 3
"fast" DEC 5000 **speedup**	2.15	2.32	2.00
"slow" DEC 5000 **slowdown**	11.67	13.49	9.00
SPARCstation 10 **speedup**	3.12	3.92	3.24

Matrix dimensions are 500.

"slow" DECstation 5000 on a set of such problems.[3] Recall that the fast DECstation does NaN arithmetic (incorrectly) at the same speed as with conventional arguments, whereas the slow DECstation computes correctly but 80 times slower. Table IV gives the speeds for both DECstations. The slow DEC 5000 goes 18 to 30 times slower than the fast DEC 5000. On some other examples, where only infinities but no NaN's occurred, the speedups ranged from 3.5 to 6.0 on both machines. Table IV also shows the speedups observed on a SPARCstation 10, where both ∞ and NaN arithmetic are implemented correctly and with full speed.

V. EIGENVECTOR COMPUTATION

We now consider another opportunity to exploit IEEE exception handling. The problem is to compute eigenvectors of general complex matrices. This example, in contrast to early ones, requires recomputing the answer slowly after an exception occurs, as in our paradigm.

Let A be an n-by-n complex matrix. If nonzero vectors v and u, and a scalar λ satisfy $Av = \lambda v$ and $u^*A = \lambda u^*$ (* denotes conjugate transpose), then λ is called an eigenvalue, and v and u^* are called the right and left eigenvectors associated with the eigenvalue λ, respectively. In LAPACK, the task of computing eigenvalues and the associated eigenvectors is performed in the following stages (as in the routine CGEEV):

[3]The test matrices together with the software can be obtained via anonymous ftp [7].

1) A is reduced to upper Hessenberg form H, which is zero below the first subdiagonal. The reduction can be written $H = Q^*AQ$ with Q unitary [12].
2) H is reduced to Schur form T. The reduction can be written $T = S^*HS$, where T is an upper triangular matrix and S is unitary [12]. The eigenvalues are on the diagonal of T.
3) CTREVC computes the eigenvectors of T. Let V be the matrix whose columns are the right eigenvectors of T. Then $S \cdot V$ are the right eigenvectors of H, and $Q \cdot S \cdot V$ are the right eigenvectors of A. Similarly, we can compute the left eigenvectors of A from those of T.

Let us first examine the important stage of calculating the eigenvectors of an upper triangular matrix T. The eigenvalues of T are $t_{11}, t_{22}, \cdots, t_{nn}$. To find a right eigenvector v associated with the eigenvalue t_{ii}, we need to solve the homogeneous equation $(T - t_{ii}I)v = 0$, which can be partitioned into the block form

$$\begin{bmatrix} T_{11} - t_{ii}I & T_{12} & T_{13} \\ \mathbf{0} & 0 & T_{23} \\ \mathbf{0} & 0 & T_{33} - t_{ii}I \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \mathbf{0}. \qquad (2)$$

By backward substitution, we have $v_3 = \mathbf{0}$, $v_2 = 1$ and v_1 satisfying the equation

$$(T_{11} - t_{ii}I)v_1 = -T_{12}. \qquad (3)$$

Therefore, the problem is reduced to solving an upper triangular system (3) of dimension $(i-1)$-by-$(i-1)$. To find all the n eigenvectors we need to solve triangular system (3) for $i = 2, \cdots, n$. Since any scalar multiple of v is also an eigenvector of T, we always expect to obtain an answer by scaling the solution vector no matter how ill-conditioned or badly scaled the triangular system (3) is. For this purpose, CTREVC calls the triangular solve routine CLATRS instead of calling the triangular solver CTRSV in the BLAS. CLATRS is a complex counterpart of SLATRS as discussed in Section III, using Algorithm 2. In most common cases, however, the scaling unnecessarily introduces overhead. We reimplemented the part of CTREVC containing the triangular solve. When solving each equation (3), we first call CTRSV and test the exception flags. If exceptions occur, then we go back to call CLATRS.

To study the efficiency of the modified CTREVC, we ran the old code and our new one on random upper triangular matrices of various sizes. We observed the speedups of from 1.49 to 1.65 on the DECstation 5000, and from 1.38 to 1.46 on the Sun 4/260. In the case of overflow, each triangular solve is invoked twice, first using CTRSV yet throwing away the solutions, and second using CLATRS. Since CTRSV is about twice as fast as CLATRS (see Section 3), the performance loss is no more than 50% when a (rare) exception occurs.

To see how the performance attained from CTREVC alone effects the performance of the whole process of computing eigenvectors of general complex matrices, we timed CTREVC in the context of CGEEV. It turns out that CTREVC amounts to about 20% of the total execution time of CGEEV. Therefore, we expect that the speed of the whole process can be increased by about 8%.

VI. Symmetric Tridiagonal Eigenvalue Problem

In this section we consider the problem of findig the eigenvalues of a real symmetric tridiagonal matrix.

Let T be an n-by-n symmetric tridiagonal matrix of the form

$$T = \begin{bmatrix} a_1 & b_1 & & & \\ b_1 & a_2 & b_2 & & \\ & b_2 & \ddots & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ & & & b_{n-1} & a_n \end{bmatrix}.$$

The *bisection* method is an accurate, inexpensive, and parallelizable procedure for calculating the eigenvalues of T. The inner loop of this method is based on an integer-valued function count(σ) of a real argument σ defined as

```
Function count($\sigma$)
$C = 0$;   $t = a_1 - \sigma$;
if ($t \leq 0$) then $C = 1$;
for $i = 2$ to $n$
    $t = a_i - \sigma - \frac{b_{i-1}^2}{t}$;
    if ($t \leq 0$) then $C = C + 1$;
endfor
return $C$;
```

Thus, the count() function counts the number of non-positive t's in the above iteration. It is known that this number equals the number of eigenvalues less than or equal to σ [12]. Suppose we wish to find the eigenvalues in $(a, b]$ using bisection. First, we evaluate count(a) and count(b). The difference of the two count values is the number of eigenvalues in $(a, b]$. Now let $\sigma = (a + b)/2$, the midpoint of the interval, and evaluate count(σ). From this we can deduce how many eigenvalues lie in each of the intervals $(a, \sigma]$ and $(\sigma, b]$. Then we recursively bisect each of the two intervals. An interval containing a single eigenvalue is bisected repeated until the eigenvalue has been determined with sufficient precision.

The division involved in the recurrence for t_i may cause division by zero or overflow. Again, to prevent the occurrence of the exceptions, a more careful scheme was first developed by W. Kahan [16] and later used in LAPACK SSTEBZ routine [2]. There, the algorithm first computes a threshold *pivmin*, which is the smallest number that can divide b_i^2 without overflow. Inside the inner loop the divisor t is compared with *pivmin* and changed to —*pivmin* if it is too close to zero. Algorithm 6 gives the details of the method.

Algorithm 6 Computes the number of eigenvalues less than or equal to σ.

```
$C = 0$;
$t = 1$;
$b_0 = 0$;
for $i = 0$ to $n - 1$
    $t = a_{i+1} - \sigma - \frac{b_{i-1}^2}{t}$;
    if ($|t| <$ pivmin) then $t =$ —pivmin;
    if ($t \leq 0$) then $C = C + 1$;
endfor
return $C$;
```

On machines with IEEE floating-point arithmetic, we may rewrite the count function as in Algorithm 7, even though $\frac{b_{i-1}^2}{t}$ may overflow. Whenever this occurs, default values $\pm\infty$ are used to continue the computation.

Algorithm 7 Computes the number of eigenvalues less than or equal to σ.

```
$C = 0$;
$t = 1$;
$b_0 = 0$;
for $i = 0$ to $n - 1$
    $t = a_{i+1} - \sigma - \frac{b_{i-1}^2}{t}$;
    $C = C +$ signbit($t$);
endfor
return $C$;
```

Signbit(x) extracts the sign bit of a floating-point number x represented in the IEEE format. The returned value is either 0 or 1 depending on whether x is positive or negative. The signbit(x) can be computed quickly by logically shifting the sign bit of x to the rightmost bit position of a register, leaving zeros in all the other bits.

The correctness of Algorithm 7 relies on the fact that arithmetic with $\pm\infty$ and signed zeros ± 0 obeys certain rules defined by the IEEE standard. The merit of Algorithm 7 is that it replaces the two explicit conditional branches with a single straight-line statement, and this makes better use of floating-point pipelines. The only hardware requirement for Algorithm 7 to attain good speed is the speed of infinity arithmetic.

On a SPARCstation IPX, where infinity arithmetic is as fast as conventional arithmetic, we measured the speed of Algorithm 6 and 7 for various matrices of sizes ranging from 100 to 1000. Algorithm 7 achieved speedups ranging from 1.20 to 1.30 over Algorithm 6. We also compared the two bisection algorithms, using Algorithm 6 and 7 as the inner loops respectively, to find all eigenvalues. We were able to get speedups ranging from 1.14 to 1.24. This is due to the dominant role of the count() function in the bisection algorithm.

We also did comparisons on a distributed memory multiprocessor—the Thinking Machines CM-5 [19]. Our CM-5 configuration contains 64 33-Mhz SPARC 2 processors, interconnected by a fat-tree network. Each processing node has 8 Mbytes of local memory. Coordination and synchronization among processing nodes are achieved via explicitly passing messages. The floating-point arithmetic on the CM-5 conforms to IEEE standard, and infinity arithmetic is as fast as conventional arithmetic. Inderjit Dhillon et al. [8] have designed a parallel bisection algorithm on the CM-5, where the whole spectrum is divided into 64 subintervals, and each processing node is responsible for finding the eigenvalues within one subinterval. A dynamic load balancing scheme is incorporated when eigenvalues are not evenly distributed.

In Table V, we report three types of speedup numbers from our experiments. T_s (*algo*) stands for the running time of the *algo* on a single node of the CM-5; T_p (*algo*) stands for the running time of the *algo* on the 64 node CM-5. Thus, $\frac{T_s(algo)}{T_p(algo)}$ represents the parallel speedup of the *algo*. The two algorithms we compared are: *LAPACK_bisect* that used Algorithm 6 to get the count value, and *IEEE_bisect* that used Algorithm 7 to get the count value. The last column demonstrates the speedup of

TABLE V
SPEEDUPS OF THE PARALLEL BISECTION ALGORITHMS ON THE CM-5

Matrix size	$\frac{T_s(LAPACK\_bisect)}{T_p(LAPACK\_bisect)}$	$\frac{T_s(IEEE\_bisect)}{T_p(IEEE\_bisect)}$	$\frac{T_p(LAPACK\_bisect)}{T_p(IEEE\_bisect)}$
192	45	53	1.47
1024	61	61	1.25
4096	53	51	1.18
16384	–	–	1.18

the parallel *IEEE_bisect* against the parallel LAPACK*_bisect*. We see the speedups attained by using IEEE arithmetic ranges from 1.18 to 1.47.

VII. SINGULAR VALUE DECOMPOSITION

In this section we discuss using exception handling to speed up the computation of the singular value decomposition of a matrix [12]. This is an important linear algebra computation, with many applications. It consists of two phases. Phase 1, reduction to bidiagonal form (i.e. nonzero on the diagonal and first superdiagonal only), costs $O(n^3)$ operations, where n is the matrix dimension. Phase 2, computing singular values of a bidiagonal matrix, costs just $O(n^2)$ operations. Nevertheless, Phase 2 can take much longer than Phase 1 on machines like the CRAY-C90 because Phase 1 is readily vectorized (or parallelized), whereas Phase 2 consists of nonlinear recurrences which run at scalar speeds. For example, when $n = 200$, Phase 1 does about $2.1 \cdot 10^7$ floating point operations at a speed of 594 Megaflops, for a time of 0.036 seconds, whereas Phase 2 does $1.6 \cdot 10^6$ floating point operations at a speed of just 6.9 Megaflops, for a time of 0.23 seconds. Phase 2 takes longer than Phase 1 up to $n \approx 1200$. So in this section we will discuss using exception handling to accelerate Phase 2. Phase 2 is implemented by a slight modification of LAPACK subroutine SBDSQR [2], which we describe below.

It suffices to consider one of the main loops in SBDSQR; the others are similar. In addition to 12 multiplies and 4 addition, there are two uses of an operation we will call rot (f, g, cs, sn, r). It takes f and g as inputs, and returns $r = (f^2 + g^2)^{1/2}$, $cs = f/r$ and g/r as outputs. This simple formula is subject to failure or inaccuracy when either f or g is greater than the square root of the overflow threshold, or when both are little larger than the square root of the underflow threshold. Therefore, SBDSQR currently does a series of tests and scalings to avoid this failure. (The difference between SBDSQR and our routine is that our routine in-lines rot and uses a slightly different and move accurate scaling algorithm.) Almost all the time, these tests indicate no scaling is needed, but it is impossible to determine this without running through the whole loop. We compare the performance of two versions of SBDSQR , one which tests and scales as above, and another, which we will call SBDSQR_UNSAFE, which just uses simple single line formulas for r, cs and sn. We tested these two routines on a CRAY Y-MP (EL/2-256). The speedups depend somewhat on the matrix. The test bidiagonal matrix A had entries of the form $A_{i,i} = \zeta^{n-i}$ and $A_{i,i+1} = A_{i,i}$, with dimensions ranging from 50 to 1000. With $\zeta = 1.0001$, most speedups were between 1.28 and 1.39, with half over 1.35. With $\zeta = 1.044$, most speedups were between 1.21 and 1.31.

VIII. LESSONS FOR SYSTEM ARCHITECTS

The most important lesson is that well-designed exception handling permits the most common cases, where *no* exceptions occur, to be implemented much more quickly. This alone makes exception handling worth implementing well.

A trickier question is how fast exception handling must be implemented. There are three speeds at issue: the speed of NaN and infinity arithmetic, the speed of testing sticky flags, and the speed of trap handling. In principle, there is no reason NaN and infinity arithmetic should not be as fast as conventional arithmetic. The examples in section 4.2 showed that a slowdown in NaN arithmetic by a factor of 80 from conventional arithmetic slows down condition estimation by a factor of 18 to 30.

Since exceptions are reasonably rare, these slowdowns generally affect only the worst case behavior of the algorithm. Depending on the application, this may or may not be important. If the worst case is important, it is important that system designers provide some method of fast exception handling, either NaN and infinity arithmetic, testing the sticky flags, or trap handling. Making all three very slow will force users to code to avoid all exceptions in the first place, the original unpleasant situation exception handling was designed to avoid.

It is particularly important to have fast exception handling in a parallel computer for the following reason. The running time of a parallel algorithm is the running time of the slowest processor, and the probability of an exception occurring on at least one processor can be p times as great as on one processor, where p is the number of processors.

IX. FUTURE WORK

The design paradigm for numerical algorithms proposed in this paper is quite general and can be used to develop other numerical algorithms. These include rewriting the BLAS routine SNRM2 to compute the Euclidean norm of a vector, and the LAPACK routine SHSEIN (which now calls SLATRS) to compute the eigenvectors of a real upper Hessenberg matrix.

In complex division, gradual underflow instead of flush to zero can guarantee a more accurate result, see [5]. This requires fast arithmetic with denormalized numbers.

Floating point parallel prefix is a useful operation for various linear algebra problems. Its robust implementation with the protection against over/underflow requires fine grained detection and handling of exceptions [6].

Our final comment concerns the trade-off between the speed of NaN and infinity arithmetic and the granularity of testing for exceptions. Our current approach uses a very large granularity, since we test for exceptions only after a complete call to STRSV. For this approach to be fast, NaN and infinity arithmetic must be fast. On the other hand, a very fine grained approach would test for exceptions inside the inner loop, and so avoid doing useless NaN and infinity arithmetic. However, such frequent testing is clearly too expensive. A compromise would be to test for exceptions after one or several complete iterations of the inner loop in STRSV. This would require reimplementing STRSV. This medium grained approach is less sensitive to the speed of NaN and infinity arithmetic. The effect of granularity on performance is worth exploration.

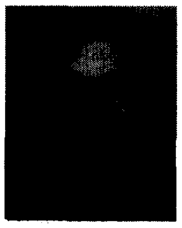
The software described in this report is available from the authors [7].

ACKNOWLEDGMENT

The authors wish to thank W. Kahan for his detailed criticism and comments. We also wish to thank Inderjit Dhillon for providing us the performance results of the bisection algorithms running on the CM-5.

REFERENCES

[1] E. Anderson, "Robust triangular solves for use in condition estimation," Comput. Sci. Dep. Tech. Rep. CS-91-142, Univ. of Tennessee, Knoxville, 1991. (LAPACK Working Note #36).
[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, Release 1.0. SIAM, Philadelphia, 1992, pp. 235.
[3] ANSI/IEEE, New York, *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 ed., 1985.
[4] ANSI/IEEE, New York. *IEEE Standard for Raix Independent Floating Point Arithmetic*, Std 854-1987 ed., 1987.
[5] J. Demmel, "Underflow and the reliability of numerical software," *SIAM J. Sci. Statist. Comput.*, vol. 5, no. 4, pp. 887–919, Dec. 1984.
[6] ——, "Specifications for robust parallel prefix operations," *Technical Report*, Thinking Machines Corp., 1992.
[7] J. Demmel and X. Li, "Faster numerical algorithms via exception handling," in *Proc. 11th Symp. Comput. Arithmetic*, M. J. Irwin, E. Swartzlander, and G. Jullien, Eds., Windsor, ON, Canada, June 29–July 2 1993. IEEE Computer Society Press, available as all.ps.Z via anonymous ftp from tr-ftp.cs.berkeley.edu, in directory pub/tech-reports/cs/csd-93-728; software is csd-93-728.shar.Z.
[8] I. S. Dhillon and J. W. Demmel, "A parallel algorithm for the symmetric tridiagonal eigenproblem and its implementation on the CM-5," in progress, 1993.
[9] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Software*, vol. 16, no. 1, pp. 1–17, Mar. 1990.
[10] J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson, "An extended set of FORTRAN basic linear algebra subroutines," *ACM Trans. Math. Software*, vol. 1, no. 1, pp. 1–17, Mar. 1988.
[11] R. L. Sites, Ed., *Alpha Architecture Reference Manual.* Burlington, MA: Digital Press, 1992.
[12] G. Bolub and C. Van Loan, *Matrix Computations*, 2nd ed. Baltimore, MD: Johns Hopkins Univ. Press, 1989.
[13] W. W. Hager, "Condition estimators," *SIAM J. Sci. Statist. Comput.*, vol. 5, pp. 311–316, 1984.
[14] N. J. Higham, "Algorithm 674: FORTRAN codes for estimating the one-norm of a real or complex matrix, with applications to condition estimation," *ACM Trans. Math. Software*, vol. 14, pp. 381–396, 1988.
[15] SPARC Internatinal Inc. *The SPARC Architecture Manual: Version 8.* Englewood Cliffs, NJ: Prentice Hall, 1992.
[16] W. Kahan, "Accurate eigenvalues of symmetric tridiagonal matrix," Computer Sci. Dep., Tech. Rep. CS41, Stanford Univ., Stanford, CA, July 1966 (revised June 1968).
[17] G. Kane, *MIPS Risc Architecture.* Englewood Cliffs, NJ: Prentice Hall, 1989.
[18] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh, "Basic linear algebra subprograms for Fortran usage," *ACM Trans. Math. Software*, vol. 5, pp. 308–323, 1979.
[19] Thinking Machines Corporation, *The Connection Machine CM-5 Technical Summary*, Cambridge MA, Oct. 1991.



James W. Demmel (M'86) received the Ph.D. degree in computer science from the University of California, Berkeley in 1983.

He is currently a Professor of Mathematics and Computer Science at the University of California, Berkeley. His research interests are in scientific computing, numerical linear algebra, and parallelism.

Dr. Demmel is co-principal investigator in the LAPACK and SCALAPACK projects, which are linear algebra libraries for high performance computers. He won the 1993 J. H. Wilkinson Prize in Numerical Analysis and Scientific Computing, the SIAM SIAG Best Linear Algebra Paper Prize in 1991 (with W. Kahan) and again in 1988, the Fox Prize in Numerical Analysis in 1986, a Presidential Young Investigator Award in 1986, and the Householder Award in 1984. He is a member of SIAM Council, the RIACS Science Council, and the Joint AMS–SIAM Committee on Applied Mathematics.

Xiaoye Li received the M.S. degree in computer science and the M.A. degree in mathematics from Penn State University, and the B.S. degree in computer science from Tsinghua University in China. She is pursuing the Ph.D. degree in computer science at the University of California, Berkeley.

She works on algorithms and software development for portable linear algebra library and scientific applications, targeted at various high performance machines. She is a member of the IEEE Computer Society and the ACM.