TRADING OFF PARALLELISM AND NUMERICAL STABILITY

J. W. DEMMEL
Computer Science Division and Mathematics Department
University of California
Berkeley, CA 94720
demmel@cs.berkeley.edu

ABSTRACT. The fastest parallel algorithm for a problem may be significantly less stable numerically than the fastest serial algorithm. We illustrate this phenomenon by a series of examples drawn from numerical linear algebra. We also show how some of these instabilities may be mitigated by better floating point arithmetic.

KEYWORDS. Parallel numerical linear algebra, numerical stability, floating point arithmetic.

Introduction

The most natural way to design a parallel numerical algorithm is to take an existing numerically stable algorithm and parallelize it. If the parallel version performs the same floating point operations as the serial version, and in the same order, one expects it to be equally stable numerically. In some cases, such as matrix operations, one expects that the parallel algorithm may reorder some operations (such as computing sums) without sacrificing numerical stability. In other cases, reordering sums could undermine stability, e.g. ODEs and PDEs.

Our purpose in this paper is to point out that designing satisfactorily fast and stable parallel numerical algorithms is not so easy as parallelizing stable serial algorithms. We identify two obstacles:

1. An algorithm which was adequate on small problems may fail once they are large enough. This becomes evident when the algorithm is used on a large parallel machine to solve larger problems than possible before. Reasons for this phenomenon include roundoff accumulation, systematically increasing condition numbers, and systematically higher probability of "random instability."

2. A fast parallel algorithm for a problem may be significantly less stable than a fast serial algorithm. In other words, there is a tradeoff between parallelism and stability.

We also discuss two techniques which sometimes remove or mitigate these obstacles. The first is *good floating point arithmetic*, which, depending on the situation, may mean carefully rounding, adequate exception handling, or the availability of extra precision without excessive slowdown. The second technique is as follows:

1. Solve the problem using a fast method, provided it is rarely unstable.
2. Quickly and reliably confirm or deny the accuracy of the computed solution. With high probability, the answer just (quickly) computed is accurate enough to keep.
3. Otherwise, recompute the desired result using a slower but more reliable algorithm.

This paradigm lets us combine a fast but occasionally unstable method with a slower, more reliable one to get guaranteed reliability and usually quick execution. One could also change the third step to just issue a warning, which would guarantee fast execution, guarantee not to return an unreliable answer, but occasionally fail to return an answer at all. Which paradigm is preferable is application dependent.

The body of the paper consists of a series of examples drawn both from the literature and from the experience in the LAPACK project [3]. As our understanding of problems improves, the status of these tradeoffs will change. For example, until recently it was possible to use a certain parallel algorithm for the symmetric tridiagonal eigenvalue problem only if the floating point arithmetic was accurate enough to simulate double the input precision [19, 35, 73, 10]. Just recently, a new formulation of the inner loop was found which made this unnecessary [48]. The fact remains that for a number of years, the only known way to use this algorithm stably was via extra precision. So one can say that the price of insufficiently accurate arithmetic was not an inability to solve this problem, but several years of lost productivity because a more straightforward algorithm could not be used.

Section 1. describes how algorithms which have been successful on small or medium sized problems can fail when they are scaled up to run on larger machines and problems. Section 2. describes parallel algorithms which are less stable than their serial counterparts. The benefit of better floating point arithmetic will be pointed out while discussing the relevant examples, and overall recommendations for arithmetic summarized in section 3.

1. Barriers to Scaling up Old Algorithms

1.1. Sparse Cholesky on the Cray Y-MP and Cray 2

We discuss the experience of Russell Carter in porting an existing code for sparse Cholesky factorization to a Cray Y-MP [15]. Cholesky is a very stable algorithm, and this code had been in use for some time. The Cray Y-MP was larger than machines previously available, and Carter ran it on a large linear system $Ax = b$ from a structural model. A

Computer	Bits	Nominal precision	Displacement
Cray 2	128	1.e-29	.447440341
Convex 220	64	1.e-16	.44744033<u>9</u>
IRIS	64	1.e-16	.44744033<u>9</u>
IBM 3090	64	1.e-17	.44744034<u>4</u>
Cray 2	64	4.e-15	.4474403<u>03</u>
Cray Y-MP	64	4.e-15	.447436<u>106</u>

Table 1: Sparse Cholesky Results

had dimension 16146. Results are shown in table 1. The first column is the computer with which the problem is solved, the second is the number of bits in the floating point format, the third column is the approximate relative accuracy with which the floating point arithmetic can represent numbers (which is not the accuracy of computation on the Cray [55]), and the last column records one of the solution components of interest. The top line, which is done to about twice the accuracy of the others, is accurate in all the digits shown. In the other results the incorrect digits are underlined.

It can be seen that the Cray Y-MP loses two more digits than the Cray 2, even though both are using 64 bit words, and their 48-fraction-bit arithmetics are quite similar. The reason for this discrepancy is that both the Cray 2 and Cray Y-MP subtract incorrectly, but the Cray 2 does so in an unbiased manner. In particular, the inner loop of Cholesky computes $a_{ii} - \sum_{j=1}^{i-1} a_{ij}^2$, where a_{ii} is positive and the final result is also positive. Whenever the Cray 2 subtracts an a_{ij}^2, the average error is 0; the computed difference is too large as often as it is too small. On the Cray Y-MP, on the other hand, the difference is always a little too big. So the error accumulates with each subtract, instead of averaging out as on the Cray 2. The accumulating error is very small, and makes little difference as long as there are not too many terms in the sum. But $n = 16146$ was finally large enough to cause a noticeable loss of 2 decimal places in the final answer. The fix used by Carter was to use the single precision iterative refinement routine `SGERFS` in LAPACK [3].

The lessons of this example are that instability may become visible only when a problem's dimension becomes large enough, and that accurate arithmetic would have mitigated the instability.

1.2. Increasing condition numbers

The last section showed how instability can arise when errors accumulate in the course of solving larger problems than ever attempted before. Another way this can arise is when the condition number of the problem grows too rapidly with its size. This may happen, for example, when we increase the mesh density with which we discretize a particular PDE. Consider the biharmonic equation $u_{xxxx} + u_{yyyy} = f$ on an n by n mesh, with boundary conditions chosen so that it represents the displacement of a square sheet fixed at the edges. The linear system $Ax = b$ resulting from the discretization has a condition number which grows like n^4. Suppose that we want to compute the solution correct to 6 decimal digits (a

relative accuracy of 10^{-6}).

Generally one can solve $Ax = b$ with a backward error of order ε, the machine precision. Write $\varepsilon = 2^{-p}$, where p is the number of bits in the floating point fraction. This means the relative accuracy of the answer will be about $\varepsilon n^4 = 2^{-p} n^4$. For this to be less than or equal to 10^{-6}, we need $2^{-p} n^4 \leq 10^{-6}$ or $p \geq 4 \log_2 n + 6 \log_2 10 \approx 4 \log_2 n + 20$. In IEEE double precision, $p = 52$ so we must have $n \leq 259$, which is fairly small.

One might object that for the biharmonic equation, Laplace's equation, and others from mathematical physics, if they have sufficiently regularity, then one can use techniques like multigrid, domain decomposition and FFTs to get accurate solutions for larger n (for the biharmonic, use boundary integral methods or [12]). This is because these methods work best when the right hand side b and solution x are both reasonably smooth functions, so that the more extreme singular values of the differential operators are not excited, and the bad conditioning is not visible. One often exploits this in practice. So in the long run, clever algorithms may become available which mitigate the ill-conditioning. In the short run, more accurate arithmetic (a larger p) would have permitted conventional algorithms to scale up to larger problems without change and remain useful longer. We will see this phenomenon later as well.

1.3. Increasing probability of random instabilities

Some numerical instabilities only occur when exact or near cancellation occurs in a numerical process. In particular, the result of the cancellation must suffer a significant loss of relative accuracy, and then propagate harmfully through the rest of the algorithm. The best known example is Gaussian elimination without pivoting, which is unstable precisely when a leading principal submatrix is singular or nearly so. The set of matrices where this occurs is defined by a set of polynomial equations: $\det(A_r) = 0$, $r = 1, ..., n$, where A_r is a leading r by r principal submatrix of the matrix A. More generally, the set of problems on or near which cancellation occurs is an *algebraic variety* in the space of the problem's parameters, i.e. defined by a set of polynomial equations in the problem's parameters. Geometrically, varieties are smooth surfaces except for possible self intersections and cusps. Other examples of such varieties include polynomials with multiple roots, matrices with multiple eigenvalues, matrices with given ranks, and so on [23, 24, 40, 41].

Since instability arises not just when our problem lies on a variety, but when it is near one, we want to know how many problems lie near a variety. One may conveniently reformulate this as a probabilistic question: given a "random" problem, what is the probability that it lies within distance η of a variety? We may choose η to correspond to an accuracy threshold, problems lying outside distance η being guaranteed to be solved accurately enough, and those within η being susceptible to significant inaccuracy. For example, we may choose $\eta = 10^d \varepsilon$ (where ε is the machine precision) if we wish to guarantee at least d significant decimal digits in the answer.

It turns out that for a given variety, we can write down a simple formula that estimates this probability as a function of several simple parameters [24, 41]: the probability per

second P of being within η of an instability is [55]

$$P = C \cdot M^k \cdot S \cdot \eta$$

where C and k are problem-dependent constants, M is the memory size in words, and S is the machine speed in flops per second.

For example, consider an SIMD machine where we assign each processor the job of LU decomposition of an independent random real matrix of fixed size n, and repeat this. We choose LU without pivoting in order to best match the SIMD architecture of the machine. We assume that each processor has an equal amount of memory, so that M is proportional to the number of processors $M = p \cdot M_p$. From [41], we use the fact that the probability that a random n by n real matrix has a condition number $\|A\|_F \|A^{-1}\|_2$ exceeding $1/\eta$ is asymptotic to $n^{3/2}\eta$. Finally, suppose that we want to compute the answer with d decimal digits of accuracy, so that we pick $\eta = 10^d \varepsilon$. Combining this information, we get that the probability per second that an instability occurs (because a matrix has condition number exceeding $1/\eta$) is at least about

$$P = p \times \frac{S}{\frac{2}{3}n^3} \times n^{3/2} 10^d \varepsilon = \frac{3}{2n^{3/2} M_p} \cdot M \cdot S \cdot 10^d \cdot \varepsilon$$

The important features of this formula is that is grows with increasing memory size M, with increasing machine speed S, and desired accuracy d, all of which are guaranteed to grow. We can lower the probability, however, by shrinking ε, i.e. by using more accurate arithmetic.

One might object that a better solution is to use QR factorization with Givens rotations instead of LU, because this is guaranteed to be stable without pivoting, and so is amenable to SIMD implementation. However, it costs three times as many flops. So we see there is a tradeoff between speed and stability.

If we instead fill up the memory with a single matrix of size $M^{1/2}$ by $M^{1/2}$, then the probability changes to $P = 1.5 \cdot M^{-3/4} \cdot S \cdot 10^d \cdot \varepsilon$. Interestingly, the probability goes down with M. The reason is that the time to solve an $M^{1/2}$ by $M^{1/2}$ matrix grows like $M^{3/2}$, so that the bigger the memory, the fewer such problems we can solve per second.

Another consequence of this formula is that random testing intended to discover instabilities in a program is more effective when done at low precision.

2. Trading Off Numerical Accuracy and Parallelism in New Algorithms

2.1. Fast BLAS

The BLAS, or Basic Linear Algebra Subroutines, are building blocks for many linear algebra codes, and so they should be as efficient as possible. We describe two ways of accelerating them that sacrifice some numerical stability to speed. The stability losses are not dramatic, and a reasonable BLAS implementation might consider using them.

Strassen's method is a fast way of doing matrix multiplication based on multiplying 2-by-2 matrices using 7 multiplies and 15 or 18 additions instead of 8 multiplies and 4 additions

[1]. Strassen reduces n by n matrix multiplication to $n/2$ by $n/2$ matrix multiplication and addition, and recursively to $n/2^k$ by $n/2^k$. Its overall complexity is therefore $O(n^{\log_2 7}) \approx O(n^{2.81})$ instead of $O(n^3)$. The constant in the $O(\cdot)$ is, however, much larger for Strassen's than for straightforward matrix multiplication, and so Strassen's is only faster for large matrices. In practice once k is large enough so the $n/2^k$ by $n/2^k$ submatrices fit in fast memory, conventional matrix multiply may be used. A drawback of Strassen's method is the need for extra storage for intermediate results. It has been implemented on the Cray 2 [9, 8] and IBM 3090 [50].

The conventional error bound for matrix multiplication is as follows:

$$|fl_{\mathrm{Conv}}(A \cdot B) - A \cdot B| \leq n \cdot \varepsilon \cdot |A| \cdot |B|$$

where the absolute values of matrices and the inequality are meant componentwise. The bound for Strassen's [13, 14, 49] is

$$\|fl_{\mathrm{Strassen}}(A \cdot B) - A \cdot B\|_M \leq f(n) \cdot \varepsilon \cdot \|A\|_M \cdot \|B\|_M + O(\varepsilon^2)$$

where $\| \cdot \|_M$ denotes the largest component in absolute value, and $f(n) = O(n^{\log_2 12}) \approx O(n^{3.6})$. This can be extended to all the other BLAS, such as triangular system solving with many right hand sides [49], as well as many methods besides Strassen's [11].

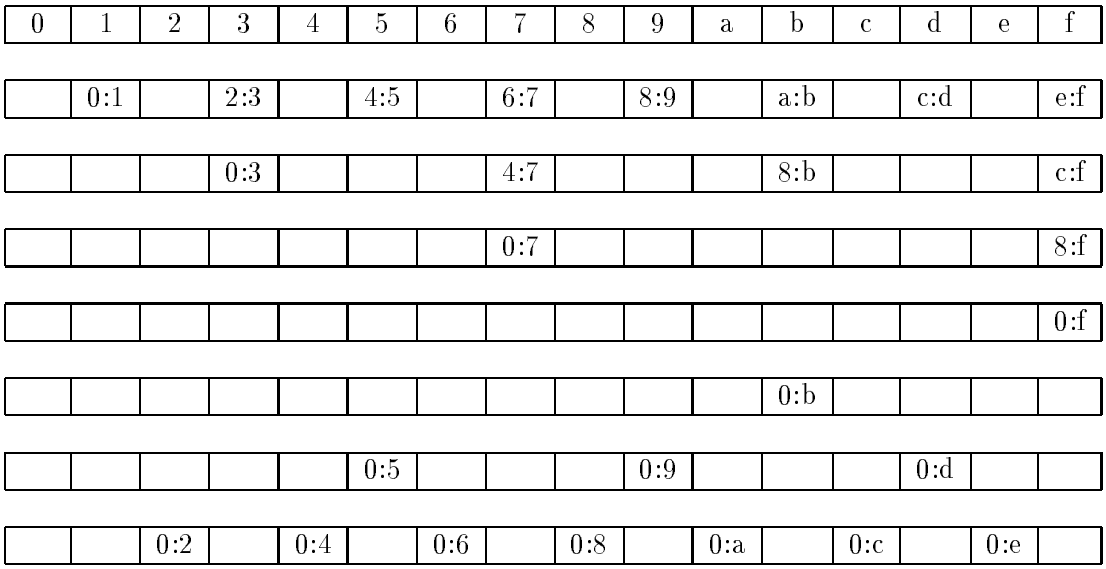
These bounds differ when there is significant difference in the scaling of A and B. For example, changing A to AD and B to $D^{-1}B$ where D is diagonal does not change the error bound for conventional multiplication, but can make Strassen's arbitrarily large. Also, if $A = |A|$ and $B = |B|$, then the conventional bound says each component of $A \cdot B$ is computed to high relative accuracy; Strassen's can not guarantee this.

On the other hand, most error analyses of Gaussian elimination and other matrix routines based on BLAS do not depend on this difference, and remain mostly the same when Strassen based BLAS are used [27]. Only when the matrix or matrices are strongly graded (the diagonal matrix D above is ill-conditioned) will the relative instability of Strassen's be noticed.

Strictly speaking, the tradeoff of speed and stability between conventional and Strassen's matrix multiplication does not depend on parallelism, but on the desire to exploit memory hierarchies in modern machines. The next algorithm, a parallel algorithm for solving triangular systems, could only be of interest in a parallel context because it uses significantly more flops than the conventional algorithm.

The algorithm may be described as follows. Let T be a unit lower triangular matrix (a nonunit diagonal can easily be scaled to be unit). For each i from 1 to $n-1$, let T_i equal the identity matrix except for column i where it matches T. Then it is simple to verify $T = T_1 T_2 \cdots T_{n-1}$ and so $T^{-1} = T_{n-1}^{-1} \cdots T_2^{-1} T_1^{-1}$. One can also easily see that T_i^{-1} equals the identity except for the subdiagonal of column i, where it is the negative of T_i. Thus T_i^{-1} comes free, and the work to be done is to compute the product $T_{n-1}^{-1} \cdots T_1^{-1}$ in $log_2 n$ parallel steps using a tree. Each parallel step involves multiplying n by n matrices (which are initially quite sparse, but fill up), and so takes about $log_2 n$ parallel substeps, for a total of $log_2^2 n$. Error analysis of this algorithm [66] yields an error bound proportional to $\kappa(T)^3 \varepsilon$ where $\kappa(T) = \|T\| \cdot \|T^{-1}\|$ is the condition number and ε is machine precision; this is in

Figure 1: Parallel Prefix on 16 Data Items



contrast to the error bound $\kappa(T)\varepsilon$ for the usual algorithm. The error bound for the parallel algorithm may be pessimistic — the worst example we have found has an error growing like $\kappa(T)^{1.5}\varepsilon$ — but shows that there is a tradeoff between parallelism and stability.

2.2. Parallel prefix

This parallel operation, also called *scan*, may be described as follows. Let $x_0, ...x_n$ be data items, and $\cdot$ any associative operation. Then the scan of these n data items yields another n data items defined by $y_0 = x_0$, $y_1 = x_0 \cdot x_1$, ... , $y_i = x_0 \cdot x_1 \cdots x_i$; thus y_i is the reduction of x_0 through x_i. The attraction of this operation, other than its usefulness, is its ease of implementation using a simple tree of processors. We illustrate in figure 1 for $n = 15$, or f in hexadecimal notation; in the figure we abbreviate x_i by i and $x_i \cdots x_j$ by $i : j$. Each row indicates the values held by the 16 processors; after the first row only the data that changes is indicated. Each updated entry combines its current value with one a fixed distance to its left.

Parallel prefix may be used to solve linear recurrence relations. For example, to evaluate $z_{i+1} = a_i z_i + b_i$, $i \geq 0$, $z_0 = 0$, we do the following operations:

Compute $p_i = a_0 \cdots a_i$ using parallel prefix multiplication
Compute $\beta_i = b_i / p_i$ in parallel
Compute $s_i = \beta_0 + \cdots + \beta_{i-1}$ using parallel prefix addition
Compute $z_i = s_i \cdot p_{i-1}$ in parallel

This approach extends to n term linear recurrences $z_{i+1} = \sum_{j=0}^{n-2} a_{i,j} z_{i-j} + b_i$, but the associative operation becomes $n - 1$ by $n - 1$ matrix multiplication. Basic linear algebra operations which can be solved this way include tridiagonal Gaussian elimination (a three

term recurrence), solving bidiagonal linear systems (two terms), Sturm sequence evaluation for the symmetric tridiagonal eigenproblem (three terms), and the bidiagonal dqds algorithm for singular values (three terms) [63].

The numerical stability of these algorithms is not completely understood. For some applications, it is easy to see the error bounds are rather worse than the those of the sequential implementation [20]. For others, such as Sturm sequence evaluation [76], empirical evidence suggests it is stable enough to use in practice.

Another source of instability besides roundoff is susceptibility to over/underflow, because of the need to compute extended products (such as $p_i = a_0 \cdots a_i$ above). These over/underflows are often unessential because the output will eventually be the solution scaled to have unit norm (inverse iteration for eigenvectors). But to use parallel prefix, one must either scale before multiplication, or deal with over/underflow after it occurs; the latter requires reasonable exception handling [25]. In the best case, a user-level trap handler would be called to deal with scaling after over/underflow, requiring no overhead if no exceptions occur. Next best is an exception flag that could be tested, provided this can also be done quickly. The worst situation occurs when all exceptions require a trap into operating system code, which is then hundreds or thousands of times slower than a single floating point operation; this is the case on the DEC α chip, for example. In this case it is probably better to code defensively by scaling every step to avoid all possibility of over/underflow. This is unfortunate because it makes portable code so hard to write: what is fastest on one machine may be very slow on another, even though both formally implement IEEE arithmetic.

2.3. Linear equation solving

In subsection 2.1., we discussed the impact of implementing LU decomposition using BLAS based on Strassen's method. In this section we discuss other variations on linear equation solving where parallelism (or just speed) and numerical stability trade off.

Parallelism in LU decomposition (and others) is often attained by blocking. For example, if A is symmetric and positive definite, its Cholesky factorization $A = R^T R$ may be divided into three blocks as follows:

$$A = R^T R = \begin{bmatrix} R_{11}^T & 0 & 0 \\ R_{12}^T & R_{22}^T & 0 \\ R_{13}^T & R_{23}^T & R_{33}^T \end{bmatrix} \cdot \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix}$$

LAPACK uses the Level 3 BLAS which perform matrix multiplication and triangular system solving in its implementation of this algorithm [3]. On some machines, solving triangular systems is rather less efficient than matrix multiplication, so that an alternative algorithm using only matrix multiplication is preferred. This can be done provided we compute the following block decomposition instead of standard Cholesky:

$$A = LU = \begin{bmatrix} I & 0 & 0 \\ L_{21} & I & 0 \\ L_{31} & L_{32} & I \end{bmatrix} \cdot \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix}$$

Pivoting Method	Pivot Search Cost (serial)	Worst Pivot Growth	Average Pivot Growth
Complete	n^2	$O(n^{1+x})$	$n^{1/2}$
Partial	n	2^{n-1}	$n^{2/3}$
Pairwise	1	4^{n-1}	$O(n)$
Parallel	1	2^{n-1}	$e^{n/4\log n}$

Table 2: Stability of various pivoting schemes in LU decomposition

In [28] it is shown that using this block LU to solve the symmetric positive definite system $Ax = b$ yields a solution $\hat{x}$ satisfying $(A+\delta A)\hat{x} = b$, with $\|\delta A\| = O(\varepsilon)(\kappa(A))^{1/2}\|A\|$, where $\kappa(A) = \|A\| \cdot \|A^{-1}\|$ is the condition number. This contrasts with the standard backward stability analysis of Cholesky which yields $\|\delta A\| = O(\varepsilon)\|A\|$. So the final error bound from block LU is $O(\varepsilon)(\kappa(A))^{3/2}$, much bigger than $O(\varepsilon)\kappa(A)$ for Cholesky. This is the price paid in stability for speed up.

Another tradeoff occurs in the choice of pivoting strategy [77]. The standard pivot strategies are complete pivoting (where we search for the largest entry in the remaining submatrix), partial pivoting (the usual choice, where we only search the current column for the largest entry), pairwise pivoting [72] (where only rows n and $n-1$ engage in pivoting and elimination, then rows $n-1$ and $n-2$ and so on up to the top) and parallel pivoting (where the remaining rows are grouped in pairs, and engage in pivoting and elimination simultaneously). Neither pairwise nor parallel pivoting require pivot search outside of two rows, but pairwise pivoting is inherently sequential in its access to rows, whereas parallel pivoting (as its name indicates) parallelizes easily. Table 2 summarizes the analysis in [77] of the speed and stability of these methods[1]. The point is that in the worst case partial, pairwise and parallel pivoting are all unstable, but on average only parallel pivoting is unstable. This is why we can using partial pivoting in practice: its worst case is very rare, but parallel pivoting is so often unstable as to be unusable. We note that an alternate kind of parallel pivoting discussed in [42] appears more stable, apparently because it eliminates entries in different columns as well as rows simultaneously. A final analysis of this problem remains to be done. We also note that, on many machines, the cost of partial pivoting is asymptotically negligible compared to the overall computation; the benefit of faster pivoting is solving smaller linear systems more efficiently.

We close by describing the fastest known parallel algorithm for solving $Ax = b$ [18]. It is also so numerically unstable as to be useless in practice. There are four steps:

1) Compute the powers of A (A^2, A^3, ... , A^{n-1}) by repeated squaring ($\log_2 n$ matrix multiplies of $\log_2 n$ steps each).
2) Compute the traces $s_i = \mathrm{tr}(A^i)$ of the powers in $\log_2 n$ steps.
3) Solve the Newton identities for the coefficients a_i of the characteristic poly-

[1]Some table entries have been proven, some are empirical with some theoretical justification, and some are purely empirical. Alan Edelman believes the $n^{2/3}$ average case pivot growth for partial pivoting should really be $n^{1/2}$.

nomial; this is a triangular system of linear equations whose matrix entries and right hand side are known integers and the s_i (we can do this in $\log_2^2 n$ steps as described above).
4) Compute the inverse using Cayley-Hamilton Theorem (in about $\log_2 n$ steps).

The algorithm is so unstable as to lose all precision in inverting $3I$ in double precision, where I is the identity matrix of size 60 or larger.

2.4. The symmetric eigenvalue problem and singular value decomposition

The basic parallel methods available for dense matrices are summarized as follows. We assume the reader is acquainted with methods discussed in [47].

1. Jacobi, which operates on the original (dense) matrix.
2. Reduction from dense to tridiagonal (or bidiagonal) form, followed by
   (a) Bisection (possibly accelerated), followed by inverse iteration for eigenvectors (if desired).
   (b) Cuppen's divide and conquer method.
   (c) QR iteration (and variations).

Jacobi has been shown to be more stable than the other methods on the list, provided it is properly implemented, and only on some classes of matrices (essentially, those whose symmetric positive definite polar factor H can be diagonally scaled as $D \cdot H \cdot D$ to be well-conditioned [30, 71]; for the SVD we use the square of the polar factor). In particular, Jacobi is capable of computing tiny eigenvalues or singular values with higher relative accuracy than methods relying on tridiagonalization. So far the error analyses of these proper implementations have depended on their use of 2-by-2 rotations, as used in conventional Jacobi. Therefore, the inner loop of these algorithms perform operations on pairs of rows or columns, i.e. Level 1 BLAS [56]. On many machines, it is more efficient to do matrix-matrix operations like level 3 BLAS [31], so one is motivated to use *block Jacobi* instead, where groups of Jacobi rotations are accumulated into a single larger orthogonal matrix, and applied to the matrix with a single matrix-matrix multiplication [67, 68, 70]. It is unknown whether this blocking destroys the subtler error analyses in [30, 71]; it is easy to show that the conventional norm-based backward stability analysis of Jacobi is not changed by blocking.

Reduction from dense to tridiagonal form is eminently parallelizable too. Having reduced to tridiagonal form, we have several parallel methods from which to choose. Bisection and QR iteration can both be reformulated as three-term linear recurrences, and so implemented using parallel prefix in $O(\log_2 n)$ time as described in section 2.2. The stability is unproven. Experiments with bisection [76] are encouraging, but the only published analysis [20] is very pessimistic. Initial results on the dqds algorithm for the bidiagonal SVD, on the other hand, indicate stability may be preserved in some cases [63]. On the other hand, bisection can easily be parallelized by having different processors refine disjoint intervals, evaluating

the Sturm sequence in the standard serial way. This involves much less communication, and is preferable in most circumstances, unless there is special support for parallel prefix.

Having used bisection to compute eigenvalues, we must use inverse iteration to compute eigenvectors. Simple inverse iteration is also easy to parallelize, with each processor independently computing the eigenvectors of the eigenvalues it owns. However, there is no guarantee of orthogonality of the computed eigenvectors, in contrast to QR iteration or Cuppen's method [53]. In particular, to achieve reasonable orthogonality one must reorthogonalize eigenvectors against those of nearby eigenvalues. This requires communication to identify nearby eigenvalues, and to transfer the eigenvectors [51]. In the serial implementation in [53], each iterate during inverse iteration is orthogonalized against previously computed eigenvectors; this is not parallelizable. The parallel version in [51] completes all the inverse iterations in parallel, and then uses modified Gram-Schmidt in a pipeline to perform the orthogonalization. To load balance, vector j was stored on processor $j \bmod p$ (p is the number of processors), and as a result reorthogonalization took a very small fraction of the total time; however, this may only have been effective because of the relatively slow floating point on the machine used (iPSC-1). In any event, the price of guaranteed orthogonality among the eigenvectors is reduced parallelism.

Cuppen's method has been analyzed by many people [19, 35, 73, 51, 54, 10, 48]. At the center of the algorithm is the solution of the *secular equation* $f(\lambda) = 0$, where f is a rational function in λ whose zeros are eigenvalues. This algorithm, while simple and attractive, proved hard to implement stably. The trouble was that to guarantee the computed eigenvectors were orthogonal, it appeared that the roots of $f(\lambda) = 0$ had to be computed in double the input precision [10, 73]. When the input is already in double precision (or whatever is the largest precision supported by the machine), then quadruple would be needed, which may be simulated using double provided double is accurate enough [22, 64]. So the availability of Cuppen's algorithm hinged on having sufficiently accurate floating point arithmetic [73, 10]. Recently, however, Gu and Eisenstat [48] have found a new way to implement this algorithm which makes extra precision unnecessary. Thus, even though carefully rounded floating point turned out not to be necessary to use Cuppen's algorithm, it took several years of research to discover this, so the price paid for poorly rounded floating point was several years of delay.

2.5. The nonsymmetric eigenproblem

Five kinds of parallel methods for the nonsymmetric eigenproblem have been investigated:

1. Hessenberg QR iteration [6, 79, 78, 21, 45, 37, 82, 81, 75],
2. Reduction to nonsymmetric tridiagonal form [46, 32, 43, 44],
3. Jacobi's method [38, 39, 74, 61, 69, 65, 80],
4. Divide and conquer based on Newton's method or homotopy continuation [16, 17, 83, 57, 58, 34]
5. Divide and conquer based on the matrix sign-function [59, 7, 60]

In contrast to the symmetric problem or SVD, no guaranteed stable and highly parallel algorithm for the nonsymmetric problem exists. Reduction to Hessenberg form (the prerequisite to methods (1) and (4) above) can be done efficiently [33, 36], but Hessenberg QR is hard to parallelize, and the other approaches are not guaranteed to converge and/or produce stable results. We summarize the tradeoffs among these methods here; for a more detailed survey, see [26].

Hessenberg QR is the serial method of choice for dense matrices. There have been a number of attempts to parallelize it, all of which maintain numerical stability since they continue to apply only orthogonal transformations to the original matrix. They instead sacrifice convergence rate or perform more flops in order to introduce higher level BLAS or parallelism. So far the parallelism has been too modest or too fine-grained to be very advantageous. In the paradigm described in the introduction, where we fall back on a slower but more stable algorithm if the fast one fails, Hessenberg QR can play the role of the stable algorithm.

Reduction to nonsymmetric tridiagonal form (followed by the tridiagonal LR algorithm) requires nonorthogonal transformations. The algorithm can break down, requiring restarting with different initial conditions [62]. Even if it does not break down, the nonorthogonal transformations required can be arbitrarily ill-conditioned, so sacrificing stability. By monitoring the condition number and restarting if it exceeds a threshold, some stability can be maintained at the cost of random running time. The more stability is demanded, the longer the running time may be, and there is no upper bound.

Jacobi's method can be implemented with orthogonal transformations only, maintaining numerical stability at the cost of linear convergence, or use nonorthogonal transformations which retain asymptotic quadratic convergence but can be arbitrarily ill-conditioned, and so possibly sacrifice stability. Orthogonal Jacobi could play the role of a slow but stable algorithm, but linear convergence makes it quite slow. The condition number of the transformation in nonorthogonal Jacobi could be monitored, and another scheme used if it is too large.

Divide and conquer using Newton or homotopy methods is applied to a Hessenberg matrix, setting the middle subdiagonal entry to zero, solving the two independent subproblems in parallel, and merging the answers of the subproblems using either Newton or a homotopy. There is parallelism in solving the independent subproblems, and in solving for the separate eigenvalues; these are the same sources of parallelism as in Cuppen's method. These methods can fail to be stable for the following reasons. Newton's method can fail to converge. Both Newton and homotopy may appear to converge to several copies of the same root without any easy way to tell if a root has been missed, or if the root really is multiple. To try to avoid this with homotopy methods requires communication to identify homotopy curves that are close together, and smaller step sizes to follow them more accurately. The subproblems produced by divide and conquer may potentially be more ill-conditioned than the original problem. See [52] for further discussion.

Divide and conquer using the matrix sign function (or a similar function) computes an orthogonal matrix $Q = [Q_1, Q_2]$ where Q_1 spans a right invariant subspace of A, and then

divides the spectrum by forming $QAQ^T = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{21} \end{bmatrix}$. To attain reasonable efficiency, Q_1 should have close to $n/2$ columns, where n is the dimension, or if the user only wants some eigenvalues, it should span the corresponding, or slightly larger, invariant subspace. One way to form Q is via the QR decomposition of the identity matrix plus the *matrix sign function* $s(A)$ of A, a function which leaves the eigenvectors alone but maps left half plane eigenvalues to -1 and right half plane eigenvalues to $+1$. A globally and asymptotically quadratically convergent iteration to compute $s(A)$ is $A_{i+1} = .5(A_i + A_i^{-1})$. This divides the spectrum into the left and right half planes; by applying this function to $A - \sigma I$ or $(A - \sigma I)^2$ or $e^{i\theta}A - \sigma I$, the spectrum can be separated along other lines.

This method can fail if the iteration fails to converge to an accurate enough approximation of $s(A)$. This will happen if some eigenvalue of A is too close to the imaginary axis (along which the iteration behaves chaotically). A symptom of this may be an intermediate A_i which is very ill-conditioned, so that A_i^{-1} is very inaccurate. It may require user input to help select the correct spectral dividing line. It can monitor its own accuracy by keeping track of the norm of the (2,1) block of QAQ^T; since the method only applies orthogonal transformations to A, it will be stable if this (2,1) block is small.

We close with some comments on finding eigenvectors, given accurate approximate eigenvalues; this is done if only a few eigenvectors are desired. The standard method is *inverse iteration*, or solving $(A - \lambda)x_{i+1} = \alpha_i x_i$ until x_i converges to an eigenvector; α_i is chosen to keep $\|x_{i+1}\| = 1$. This involves triangular system solving with a very ill-conditioned matrix, the more so to the extent that λ is an accurate eigenvalue. This ill-conditioning makes overflow a reasonable possibility, even though we only want the scaled unit vector at the end. This means the code is to compute the answer despite possible overflow, since this overflow does not mean that the eigenvector is ill-posed or even ill-conditioned. To do this portably currently requires a "paranoid" coding style, with testing and scaling in the inner loop of the triangular solve [2], making it impossible to use machine optimized BLAS. If one could defer the handling of overflow exceptions, it would be possible to run the fast BLAS, and only redo the computation with relatively slow scaling when necessary. This is an example of the paradigm of the introduction. IEEE standard floating point arithmetic [5] provides this facility in principle. However, if exception handling is too expensive (on the DEC α chip, ∞ arithmetic requires a trap into the operating system, which is quite slow), overflow can cause a slowdown of several orders of magnitude.

For the generalized nonsymmetric eigenproblem $A - \lambda B$ we do not even know how to perform generalized Hessenberg reduction using more than the Level 1 BLAS. The sign-function and related techniques [60, 7] promise to be helpful here.

3. Recommendations for Floating Point Arithmetic

We summarize the recommendations we have made in previous sections regarding floating point arithmetic support to mitigate the tradeoff between parallelism (or speed) and stability: accurate rounding, support for higher precision, and efficient exception handling. The IEEE floating point standard [5], *efficiently implemented*, is a good model. We emphasize

the efficiency of implementation because if it is very expensive to exercise the features we need, it defeats the purpose of using them to accelerate computation.

Accurate rounding attenuates or eliminates roundoff accumulation in long sums as described in section 1.1. It also permits us to simulate higher precision cheaply, which often makes it easier to design stable algorithms quickly (even though a stable algorithm which does not rely on higher precision may exist, it may take a while to discover). This was the case for Cuppen's method (section 2.4.), and also for many of the routines for 2-by-2 and 4-by-4 matrix problems in the inner loops of various LAPACK routines, such as slasv2, which computes the SVD of a 2-by-2 triangular matrix [3, 29]. Higher precision also makes it possible to extend the life of codes designed to work on smaller problems, as they are scaled to work on larger ones with larger condition numbers (section 1.2.), or with more random instabilities (section 1.3.). It is important that the extra precision be as accurate as the basic precision, because otherwise promoting a code to higher precision can introduce bugs where none were before. A simple example is that $\arccos(x/(x^2 + y^2)^{1/2})$ can fail because the argument of arccos can exceed 1 if rounding is inaccurate in division or square root [15]. Extra range and precision are very useful, since they permit us us to forego some testing and scaling to avoid over/underflow in common computations such as $\sqrt{\sum_i x_i^2}$.

Efficient exception handling permits us to run fast "risky" algorithms which usually work, without fear of having program execution terminated. Indeed, in some cases such as condition estimation, overflow permits us to finish early (in this case overflow implies that 0 is an excellent approximate reciprocal condition number). In particular, it lets us use optimized BLAS, thereby taking advantage of the manufacturer's effort in writing them (see section 2.5.). In analogy to the argument for using RISC ("reduced instruction set computers"), we want algorithms where the most common case — no exceptions — runs as quickly as possible.

This is not useful if the price of exception handling is too high; we need to be able to run with ∞ and NaN (Not a Number) arithmetic at nearly full floating point speed. The reason is that once created, an ∞ or NaN propagates through the computation, creating many more ∞'s or NaN's. This means, for example, that the DEC α implementation of this arithmetic, which uses traps to the operating system, is too unacceptably slow to be useful. The LAPACK 2 project will produce codes assuming reasonably efficient exception handling, since this is the most common kind of implementation [4].

Acknowledgements

The author acknowledges the support of NSF grant ASC-9005933, NSF PYI grant CCR-9196022, DARPA grant DAAL03-91-C-0047 via subcontract ORA-4466.02 from the University of Tennessee, and DARPA grant DM28E04120 via subcontract W-31-109-ENG-38 from Argonne National Laboratory. He also thanks W. Kahan for numerous comments on an earlier draft of this paper.

References

[1] A. Aho, J. Hopcroft, and J Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[2] E. Anderson. Robust triangular solves for use in condition estimation. Computer Science Dept. Technical Report CS-91-142, University of Tennessee, Knoxville, 1991. (LAPACK Working Note #36).

[3] 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.

[4] E. Anderson, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, S. Hammarling, and W. Kahan. Prospectus for an extension to LAPACK: a portable linear algebra library for high-performance computers. Computer Science Dept. Technical Report CS-90-118, University of Tennessee, Knoxville, 1990. (LAPACK Working Note #26).

[5] ANSI/IEEE, New York. *IEEE Standard for Binary Floating Point Arithmetic*, Std 754-1985 edition, 1985.

[6] Z. Bai and J. Demmel. On a block implementation of Hessenberg multishift QR iteration. *International Journal of High Speed Computing*, 1(1):97–112, 1989. (also LAPACK Working Note #8).

[7] Z. Bai and J. Demmel. Design of a parallel nonsymmetric eigenroutine toolbox. Computer Science Dept. preprint, University of California, Berkeley, CA, 1992.

[8] D. H. Bailey. Extra high speed matrix multiplication on the Cray-2. *SIAM J. Sci. Stat. Comput.*, 9:603–607, 1988.

[9] D. H. Bailey, K. Lee, and H. D. Simon. Using Strassen's algorithm to accelerate the solution of linear systems. *J. Supercomputing*, 4:97–371, 1991.

[10] J. Barlow. Error analysis of update methods for the symmetric eigenvalue problem. to appear in SIAM J. Mat. Anal. Appl. Tech Reprot CS-91-21, Computer Science Department, Penn State University, August 1991.

[11] D. Bini and D. Lotti. Stability of fast algorithms for matrix multiplication. *Num. Math.*, 36:63–72, 1980.

[12] P. Bjorstad. *Numerical solution of the biharmonic equation.* PhD thesis, Stanford University, 1980.

[13] R. P. Brent. Algorithms for matrix multiplication. Computer Science Dept. Report CS 157, Stanford University, 1970.

[14] R. P. Brent. Error analysis of algorithms for matrix multiplication and triangular decomposition using Winograd's identity. *Num. Math*, 16:145–156, 1970.

[15] R. Carter. Cray Y-MP floating point and Choleksy decomposition. to appear in Int. J. High Speed Computing, 1992.

[16] M. Chu. A note on the homotopy method for linear algebraic eigenvalue problems. *Lin. Alg. Appl*, 105:225–236, 1988.

[17] M. Chu, T.-Y. Li, and T. Sauer. Homotopy method for general λ-matrix problems. *SIAM J. Mat. Anal. Appl.*, 9(4):528–536, 1988.

[18] L. Csanky. Fast parallel matrix inversion algorithms. *SIAM J. Comput.*, 5:618–623, 1977.

[19] J.J.M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[20] Kuck D. and A. Sameh. A parallel QR algorithm for symmetric tridiagonal matrices. *IEEE Trans. Computers*, C-26(2), 1977.

[21] G. Davis, R. Funderlic, and G. Geist. A hypercubde implementation of the implicit double shift QR algorithm. In *Hypercube Multiprocessors 1987*, pages 619–626, Philadelphia, PA, 1987. SIAM.

[22] T. Dekker. A floating point technique for extending the available precision. *Num. Math.*, 18:224–242, 1971.

[23] J. Demmel. On condition numbers and the distance to the nearest ill-posed problem. *Num. Math.*, 51(3):251–289, July 1987.

[24] J. Demmel. The probability that a numerical analysis problem is difficult. *Math. Comput.*, 50(182):449–480, April 1988.

[25] J. Demmel. The inherent inaccuracy of implicit tridiagonal QR. Technical report, IMA, University of Minnesota, 1992.

[26] J. Demmel, M. Heath, and H. van der Vorst. Parallel numerical linear algebra. In A. Iserles, editor, *Acta Numerica, volume 2*. Cambridge University Press, 1993 (to appear).

[27] J. Demmel and N. J. Higham. Stability of block algorithms with fast Level 3 BLAS. to appear in *ACM Trans. Math. Soft.*

[28] J. Demmel, N. J. Higham, and R. Schreiber. Block LU factorization. in preparation.

[29] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 11(5):873–912, September 1990.

[30] J. Demmel and K. Veselić. Jacobi's method is more accurate than QR. Computer Science Dept. Technical Report 468, Courant Institute, New York, NY, October 1989. (also LAPACK Working Note #15), to appear in SIAM J. Mat. Anal. Appl.

[31] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[32] J. Dongarra, G. A. Geist, and C. Romine. Computing the eigenvalues and eigenvectors of a general matrix by reduction to tridiagonal form. Technical Report ONRL/TM-11669, Oak Ridge National Laboratory, 1990. to appear in ACM TOMS.

[33] J. Dongarra, S. Hammarling, and D. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *JCAM*, 27:215–227, 1989. (LAPACK Working Note #2).

[34] J. Dongarra and M. Sidani. A parallel algorithm for the non-symmetric eigenvalue problem. Computer Science Dept. Technical Report CS-91-137, University of Tennessee, Knoxville, TN, 1991.

[35] J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenproblem. *SIAM J. Sci. Stat. Comput.*, 8(2):139–154, March 1987.

[36] J. Dongarra and R. van de Geign. Reduction to condensed form for the eigenvalue problem on distributed memory computers. Computer Science Dept. Technical Report CS-91-130, University of Tennessee, Knoxville, 1991. (LAPACK Working Note #30), to appear in Parallel Computing.

[37] A. Dubrulle. The multishift QR algorithm: is it worth the trouble? Palo Alto Scientific Center Report G320-3558x, IBM Corp., 1530 Page Mill Road, Palo Alto, CA 94304, 1991.

[38] P. Eberlein. A Jacobi method for the automatic computation of eigenvalues and eigenvectors of an arbitrary matrix. *J. SIAM*, 10:74–88, 1962.

[39] P. Eberlein. On the Schur decomposition of a matrix for parallel computation. *IEEE Trans. Comput.*, 36:167–174, 1987.

[40] A. Edelman. Eigenvalues and condition numbers of random matrices. *SIAM J. on Mat. Anal. Appl.*, 9(4):543–560, October 1988.

[41] A. Edelman. On the distribution of a scaled condition number. *Math. Comp.*, 58(197):185–190, 1992.

[42] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32:54–135, 1990.

[43] G. A. Geist. Parallel tridiagonalization of a general matrix using distributed memory multiprocessors. In *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 29–35, Philadelphia, PA, 1990. SIAM.

[44] G. A. Geist. Reduction of a general matrix to tridiagonal form. *SIAM J. Mat. Anal. Appl.*, 12(2):362–373, 1991.

[45] G. A. Geist and G. J. Davis. Finding eigenvalues and eigenvectors of unsymmetric matrices using a distributed memory multiprocessor. *Parallel Computing*, 13(2):199–209, 1990.

[46] G. A. Geist, A. Lu, and E. Wachspress. Stabilized reduction of an arbitrary matrix to tridiagonal form. Technical Report ONRL/TM-11089, Oak Ridge National Laboratory, 1989.

[47] G. Golub and C. Van Loan. *Matrix Computations*. Johns Hopkins University Press, Baltimore, MD, 2nd edition, 1989.

[48] Ming Gu and S. Eisenstat. A stable and efficient algorithm for the rank-1 modification of the symmetric eigenproblem. Computer Science Dept. Report YALEU/DCS/RR-916, Yale University, August 1992.

[49] N. J. Higham. Exploiting fast matrix multiplication within the Level 3 BLAS. *ACM Trans. Math. Soft.*, 16:352–368, 1990.

[50] IBM. *Engineering and Scientific Subroutine Library, Guide and Reference, Release 3, Program 5668-863*, 4 edition, 1988.

[51] I. Ipsen and E. Jessup. Solving the symmetric tridiagonal eigenvalue problem on the hypercube. *SIAM J. Sci. Stat. Comput.*, 11(2):203–230, 1990.

[52] E. Jessup. A case against a divide and conquer approach to the nonsymmetric eigenproblem. Technical Report ONRL/TM-11903, Oak Ridge National Laboratory, 1991.

[53] E. Jessup and I. Ipsen. Improving the accuracy of inverse iteration. *SIAM J. Sci. Stat. Comput.*, 13(2):550–572, 1992.

[54] E. Jessup and D Sorensen. A divide and conquer algorithm for computing the singular value decomposition of a matrix. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 61–66, Philadelphia, PA, 1989. SIAM.

[55] W. Kahan. How Cray's arithmetic hurts scientific computing. Presented to Cray User Group Meeting, Toronto, April 10, 1991.

[56] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[57] T.-Y. Li and Z. Zeng. Homotopy-determinant algorithm for solving nonsymmetric eigenvalue problems. to appear in Math. Comp.

[58] T.-Y. Li, Z. Zeng, and L. Cong. Solving eigenvalue problems of nonsymmetric matrices with real homotopies. *SIAM J. Num. Anal.*, 29(1):229–248, 1992.

[59] C-C. Lin and E. Zmijewski. A parallel algorithm for computing the eigenvalues of an unsymmetric matrix on an SIMD mesh of processors. Department of Computer Science TRCS 91-15, University of California, Santa Barbara, CA, July 1991.

[60] A. N. Malyshev. Parallel aspects of some spectral problems in linear algebra. Dept. of Numerical Mathematics Report NM-R9113, Centre for Mathematics and Computer Science, Amsterdam, July 1991.

[61] M.H.C. Pardekooper. A quadratically convergent parallel Jacobi process for diagonally dominant matrices with distinct eigenvalues. *J. Comput. Appl. Math.*, 27:3–16, 1989.

[62] B. Parlett. Reduction to tridiagonal form and minimal realizations. *SIAM J. Mat. Anal. Appl.*, 13(2):567–593, 1992.

[63] B. Parlett and V. Fernando. Accurate singular values and differential QD algorithms. Math Department PAM-554, University of California, Berkeley, CA, July 1992.

[64] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th Symposium on Computer Arithmetic*, pages 132–145, Grenoble, France, June 26-28 1991. IEEE Computer Society Press.

[65] A. Sameh. On Jacobi and Jacobi-like algorithms for a parallel computer. *Math. Comp.*, 25:579–590, 1971.

[66] A. Sameh and R. Brent. Solving trangular systems on a parallel computer. *SIAM J. Num. Anal.*, 14:1101–1113, 1977.

[67] R. Schreiber. Solving eigenvalue and singular value problems on an undersized systolic array. *SIAM J. Sci. Stat. Comput.*, 7:441–451, 1986. first block Jacobi reference?

[68] R. Schreiber. Block algorithms for parallel machines. In M. Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures – IMA Volumes in Mathmatics and its Applications, v. 13.* Springer-Verlag, 1988.

[69] G. Shroff. A parallel algorithm for the eigenvalues and eigenvectors of a general complex matrix. *Num. Math.*, 58:779–805, 1991.

[70] G. Shroff and R. Schreiber. On the convergence of the cyclic Jacobi method for parallel block orderings. *SIAM J. Mat. Anal. Appl.*, 10:326–346, 1989.

[71] I. Slapničar. *Accurate symmetric eigenreduction by a Jacobi method.* PhD thesis, Fernuniversität - Hagen, Hagen, Germany, 1992.

[72] D. Sorensen. Analysis of pairwise pivoting in Gaussian elimination. *IEEE Trans. Comput.*, 34:274–278, 1984.

[73] D. Sorensen and P. Tang. On the orthogonality of eigenvectors computed by divide-and-conquer techniques. *SIAM J. Num. Anal.*, 28(6):1752–1775, 1991.

[74] G. W. Stewart. A Jacobi-like algorithm for computing the Schur decomposition of a non-Hermitian matrix. *SIAM J. Sci. Stat. Comput.*, 6:853–864, 1985.

[75] G. W. Stewart. A parallel implementation of the QR algorithm. *Parallel Computing*, 5:187–196, 1987.

[76] P. Swarztrauber. A parallel algorithm for computing the eigenvalues of a symmetric tridiagonal matrix. *Math. Comp.*, 1992. to appear.

[77] L. Trefethen and R. Schreiber. Average case analysis of Gaussian elimination. *SIAM J. Mat. Anal. Appl.*, 11(3):335–360, 1990.

[78] R. van de Geijn and D. Hudson. Efficient parallel implementation of the nonsymmetric QR algorithm. In J. Gustafson, editor, *Hypercube Concurrent Computers and Applications*. ACM, 1989.

[79] Robert van de Geijn. *Implementing the QR Algorithm on an Array of Processors*. PhD thesis, University of Maryland, College Park, August 1987. Computer Science Department Report TR-1897.

[80] K. Veselić. A quadratically convergent Jacobi-like method for real matrices with complex conjugate eigenvalues. *Num. Math.*, 33:425–435, 1979.

[81] D. Watkins. Shifting strategies for the parallel QR algorithm. Dept. of pure and applied math. report, Washington State Univ., Pullman, WA, 1992.

[82] D. Watkins and L. Elsner. Convergence of algorithms of decomposition type for the eigenvalue problem. *Lin. Alg. Appl.*, 143:19–47, 1991.

[83] Zhonggang Zeng. *Homotopy-determinant algorithm for solving matrix eigenvalue problems and its parallelizations*. PhD thesis, Michigan State University, 1991.