# The Spectral Decomposition of Nonsymmetric Matrices on Distributed Memory Parallel Computers

Z. Bai* J. Demmel† J. Dongarra‡ A. Petitet§ H. Robinson¶ K. Stanley‖

October 9, 1995

**Abstract**

The implementation and performance of a class of divide-and-conquer algorithms for computing the spectral decomposition of nonsymmetric matrices on distributed memory parallel computers are studied in this paper. After presenting a general framework, we focus on a spectral divide-and-conquer (SDC) algorithm with Newton iteration. Although the algorithm requires several times as many floating point operations as the best serial QR algorithm, it can be simply constructed from a small set of highly parallelizable matrix building blocks within Level 3 BLAS. Efficient implementations of these building blocks are available on a wide range of machines. In some ill-conditioned cases, the algorithm may lose numerical stability, but this can easily be detected and compensated for.

The algorithm reached 31% efficiency with respect to the underlying PUMMA matrix multiplication and 82% efficiency with respect to the underlying ScaLAPACK matrix inversion on a 256 processor Intel Touchstone Delta system, and 41% efficiency with respect to the matrix multiplication in CMSSL on a 32 node Thinking Machines CM-5 with vector units. Our performance model predicts the performance reasonably accurately.

To take advantage of the geometric nature of SDC algorithms, we have designed a graphical user interface to let user choose the spectral decomposition according to specified regions in the complex plane.

## 1 Introduction

A standard technique in parallel computing is to build algorithms from existing high performance building blocks. For example, LAPACK [1] is built on Basic Linear Algebra Subroutines (BLAS), for which efficient implementations are available on many workstations, vector processors, and shared memory parallel machines. The recently released ScaLAPACK 1.0

*Department of Mathematics, University of Kentucky, Lexington, KY 40506.

†Computer Science Division and Mathematics Department, University of California, Berkeley, CA 94720.

‡Department of Computer Science, University of Tennessee, Knoxville, TN 37996 and Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831.

§Department of Computer Science, University of Tennessee, Knoxville, TN 37996.

¶Department of Mathematics, University of California, Berkeley, CA 94720.

‖Computer Science Division, University of California, Berkeley, CA 94720.

linear algebra library [11] is written in terms of the Parallel Block BLAS (PB-BLAS) [12], Basic Linear Algebra Communication Subroutines (BLACS) [19], BLAS and LAPACK. ScaLAPACK includes routines for LU, QR and Cholesky factorizations, and matrix inversion, and has been ported to the Intel Gamma, Delta and Paragon, Thinking Machines CM-5, and PVM clusters. The Connection Machine Scientific Software Library (CMSSL) provides analogous functionality and high performance for the CM-5.

In this paper, we use these high performance kernels to implement a spectral divide-and-conquer (SDC) algorithm for finding eigenvalues and invariant subspaces of nonsymmetric matrices on distributed memory parallel computers. The algorithm recursively divides the matrix into smaller submatrices, each of which has a subset of the original eigenvalues as its own [5, 28, 3]. On a 256 processor Intel Touchstone Delta system, the SDC algorithm reached 31% efficiency with respect to the underlying matrix multiplication (PUMMA [13]) for matrices of order 4000. and 82% efficiency with respect to the underlying ScaLAPACK 1.0 matrix inversion. On a 32 processor Thinking Machines CM-5 with vector units, a varition of the SDC algorithm obtained 41% efficiency with respect to matrix multiplication from CMSSL 3.2 for matrices of order 2048.

The nonsymmetric spectral decomposition problem has until recently resisted attempts at parallelization [16]. The conventional serial method is to use the QR algorithm [1]. The algorithm had appeared to required fine grain parallelism and be difficult to parallelize. But recently Henry and van de Geijn [21] have shown that the Hessenberg QR iteration phase can be effectively parallelized for distributed memory parallel computers with up to 100 processors. Although it does not appear to be as scalable as the algorithm presented in this paper, it may be faster on a wide range of distributed memory parallel computers. The SDC algorithm performs several times as many floating point operations as the QR algorithm, but they are nearly all within Level 3 BLAS, whereas implementations of the QR algorithm performing the fewest floating point operations use less efficient Level 1 and 2 BLAS. A thorough comparison of these algorithms will be the subject of a future paper. We also note that the algorithm discussed in this paper may be less stable than the QR algorithm in a number of circumstances. Fortunately, it is easy to detect and compensate for this loss of stability. Compared with other approaches, we believe that the new algorithm offers an effective tradeoff between parallelizability and stability.

Other parallel algorithms for the eigenproblem include Hessenberg divide-and-conquer using either Newton's method [18] or homotopies [26], and Jacobi's method [33, 32]. All these methods suffer from the use of fine-grain parallelism, instability, slow or misconvergence in the presence of clustered eigenvalues of the original problem or some constructed subproblems [16]. The other algorithms most closely related to the approach used here may be found in [2, 6, 24], where symmetric matrices, or more generally matrices with real spectra, are treated.

One of the notable features of the SDC algorithm is that it can calculate just those eigenvalues (and the corresponding invariant subspace) in a user-specified region of the complex plane. To help the user specify this region, we developed a graphical user interface for the algorithm.

The rest of this paper is organized as follows. In §2, we first present a general framework of SDC algorithms, and then focus on an SDC algorithm with Newton iteration. We show how to divide the spectrum along arbitrary circles and lines in the complex plane. The implementation and performance on Intel Delta and CM-5 are presented in §3. §4 presents a model for performance analysis, and demonstrates that it can predict the execution time reasonably accurately. §5 describes the design of an X-window user interface. Concluding remarks are given in §6.

## 2 Spectral Divide and Conquer Algorithms

### 2.1 General Framework

A general framework of SDC algorithms can be described as the following. Let

$$A = X \begin{pmatrix} J_+ & 0 \\ 0 & J_- \end{pmatrix} X^{-1} \tag{1}$$

be the Jordan canonical form of an $n \times n$ matrix $A$, where the eigenvalues of $J_+$ are the eigenvalues of $A$ inside a selected region $\mathcal{D}$ in the complex plane, and the eigenvalues of $J_-$ are the eigenvalues of $A$ outside $\mathcal{D}$. We assume that there are no eigenvalues of $A$ on the boundary of $\mathcal{D}$, otherwise we reselect or move the region $\mathcal{D}$ slightly. The invariant subspace of the matrix $A$ corresponding to the eigenvalues inside $\mathcal{D}$ are spanned by the first $l$ columns of $X$, where $l$ is the number of eigenvalues inside $\mathcal{D}$. The matrix

$$P_+ = X \begin{pmatrix} I & 0 \\ 0 & 0 \end{pmatrix} X^{-1} \tag{2}$$

is the corresponding spectral projector. Let $P_+ = QR\Pi$ be the rank revealing QR decomposition of the matrix $P_+$, where $Q$ is unitary, $R$ is upper triangular, and $\Pi$ is a permutation matrix chosen so that the leading $l$ columns of $Q$ span the range space of $P_+$. Then $Q$ yields the desired spectral decomposition:

$$Q^H AQ = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \tag{3}$$

where the eigenvalues of $A_{11}$ are the eigenvalues of $A$ inside $\mathcal{D}$, and the eigenvalues of $A_{22}$ are the eigenvalues of $A$ outside $\mathcal{D}$. By substituting the complementary projector $I - P_+$ for $P_+$ in (2), $A_{11}$ will have the eigenvalues outside $\mathcal{D}$ and $A_{22}$ will have the eigenvalues inside $\mathcal{D}$.

The crux of an SDC algorithm is to efficiently compute the desired spectral projector $P_+$ without computing the Jordan canonical form.

### 2.2 An SDC algorithm with Newton iteration

One of ways to compute the spectral projector $P_+$ is to use the matrix sign function. The matrix sign function was introduced by Roberts [31] for solving the algebraic Riccati equation. However, it was soon extended to solving the spectral decomposition problem [5]. More recent studies may be found in [28, 3].

The matrix sign function, $\mathrm{sign}(A)$, of a matrix $A$ with no eigenvalues on the imaginary axis can be defined via the Jordan canonical form of $A$ (1), where the eigenvalues of $J_+$ are in the open right half plane $\mathcal{D}$, and the eigenvalues of $J_-$ are in the open left half plane $\overline{\mathcal{D}}$. Then $\text{sign}(A)$ is

$$\text{sign}(A) \equiv X \begin{pmatrix} I & 0 \\ 0 & -I \end{pmatrix} X^{-1}.$$

It is easy to see that the matrix

$$P_+ = \frac{1}{2}(I + \text{sign}(A)) \tag{4}$$

is the spectral projector onto the invariant subspace corresponding to the eigenvalues of $A$ in $\mathcal{D}$. $l = \text{trace}(P_+) = \text{rank}(P_+)$ is the number of the eigenvalues of $A$ in $\mathcal{D}$. $I - P_+ = P_- = \frac{1}{2}(I - \text{sign}(A))$ is the spectral projector corresponding to the eigenvalues of $A$ in $\overline{\mathcal{D}}$. Let $P_+ = QR\Pi$ be the rank revealing QR decomposition of $P_+$. Then $Q$ yields the desired spectral decomposition (3), where the eigenvalues of $A_{11}$ are the eigenvalues of $A$ in $\mathcal{D}$, and the eigenvalues of $A_{22}$ are the eigenvalues of $A$ in $\overline{\mathcal{D}}$.

Since $\text{sign}(A)$ satisfies the matrix equation $(\text{sign}(A))^2 = I$, we can use Newton's method to solve this matrix equation and derive the following Newton iteration:

$$A_{j+1} = \frac{1}{2}(A_j + A_j^{-1}), \quad \text{for} \quad j = 0, 1, 2, \ldots \quad \text{with} \quad A_0 = A. \tag{5}$$

It can be shown that the iteration is globally and ultimately quadratically convergent with $\lim_{j\to\infty} A_j = \text{sign}(A)$, provided $A$ has no pure imaginary eigenvalues [31, 23]. The iteration fails otherwise. In finite precision arithmetic, the iteration could converge slowly or not at all if $A$ is "close" to having pure imaginary eigenvalues.

There are many ways to improve the accuracy and convergence rate of this basic iteration [7, 22, 25]. For example, if $\|A^2 - I\| < 1$, we may use the Newton-Schulz iteration

$$A_{j+1} = \frac{1}{2}A_j(3I - A_j^2) \quad \text{for} \quad j = 0, 1, 2, \ldots \quad \text{with} \quad A_0 = A. \tag{6}$$

to avoid the use of the matrix inverse. Although it requires twice as many floating point operations, it is more efficient whenever matrix multiply is at least twice as efficient as matrix inversion. The Newton-Schulz iteration is also quadratically convergent provided that $\|A^2 - I\| < 1$. A hybrid iteration might begin with Newton iteration until $\|A_i^2 - I\| < 1$ and then switch to Newton-Schulz iteration.

The following is an SDC algorithm with Newton iteration to compute the spectral decomposition along the pure imaginary axis.

The SDC Algorithm with Newton Iteration

> Let $A_0 = A$
> For $j = 0, 1, \ldots$ until convergence or $j > j_{\max}$ do
>     $A_{j+1} = \frac{1}{2}(A_j + A_j^{-1})$
>     if $\|A_{j+1} - A_j\|_1 \le \tau \|A_j\|_1$, exit
> End for;
> Compute $\frac{1}{2}(A_{j+1} + I) = QR\Pi$ (rank revealing QR decomposition)

$$\text{Calculate } Q^H A Q = \begin{array}{c c} & \begin{array}{cc} l & n-1 \end{array} \\ \begin{array}{c} l \\ n-1 \end{array} & \begin{pmatrix} A_{11} & A_{12} \\ E_{21} & A_{22} \end{pmatrix} \end{array}$$

Compute $\|E_{21}\|_1/\|A\|_1$ for stability test

Here $\tau$ is the stopping criterion for the Newton iteration (say, $\tau = n\varepsilon$, where $\varepsilon$ is the machine precision), and $j_{\max}$ limits the maximum number of iterations (say $j_{\max} = 40$). $l$ is the rank of $R$. On return, the generally nonzero quantity $\|E_{21}\|_1/\|A\|_1$ measures the backward stability of the computed decomposition, since by setting $E_{21}$ to zero and so decoupling the problem into $A_{11}$ and $A_{22}$, a backward error of $\|E_{21}\|_1/\|A\|_1$ is introduced. For simplicity, we use the QR decomposition with column pivoting for rank revealing, although more sophisticated rank-revealing schemes exist [10, 20].

All variations of the Newton iteration with global convergence need to compute the inverse of a matrix explicitly in one form or another. Dealing with ill-conditioned matrices and instability in the Newton iteration for computing the matrix sign function and the subsequent spectral decomposition have been studied in [3, 8]. Recently, an inverse free method for achieving better numerical stability has been proposed in [30, 4]. The advantage of the inverse free approach is obtained at the cost of more storage and arithmetic.

Since 1) any SDC algorithm could suffer numerical instability when some eigenvalues are very close to the boundary of the selected region, 2) Newton iteration is faster but somewhat less stable than the inverse free approach, and 3) testing stability is easy, we propose to use the following 3 step hybrid algorithm for a general purpose program:

1. Use the SDC algorithm with Newton iteration. If it succeeds, stop.
2. Otherwise, divide the spectrum with the inverse free method. If it succeeds, stop.
3. Otherwise, use the QR algorithm.

This 3-step approach works by trying the fastest but least stable method first, falling back to slower but more stable methods only if necessary. The same paradigm is also used in other parallel algorithms [15]. If a fast parallel version of the QR algorithm [21] becomes available, it would probably be faster than the inverse free algorithm and hence would obviate the need for the second step listed above. But the inverse free method would still be of interest if only a subset of the spectrum is desired (the QR algorithm necessarily computes the entire spectrum), or for the generalized eigenproblem of a matrix pencil $A - \lambda B$ [4].

### 2.3 Spectral Transformation

Although the SDC algorithm with Newton iteration only divides the spectrum along the pure imaginary axis, we can use Möbius and other simple transformations of the input matrix $A$ to divide along other more general curves. As a result, we can compute the eigenvalues (and corresponding invariant subspace) inside any region defined as the intersection of regions defined by these curves. This is one of major attractions of this kind of algorithms. Specifically, with Möbius transformation

$$\frac{\alpha z + \beta}{\gamma z + \delta},$$

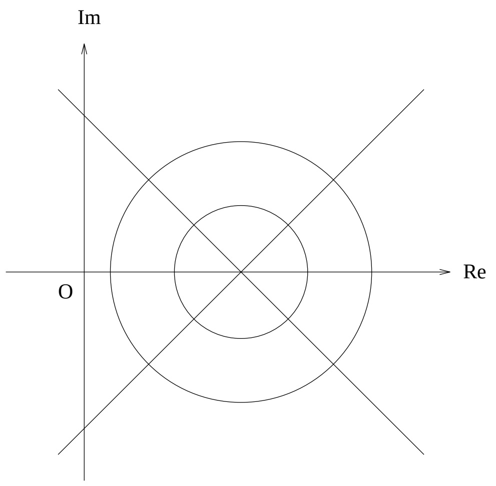


Figure 1: Different Geometric Regions for the Spectral Decomposition

where $\alpha, \beta, \gamma, \delta$ are constants, and $z$ is a complex variable, the eigenproblem $Ax = \lambda x$ is transformed to

$$(\alpha A + \beta I)x = \frac{\alpha\lambda + \beta}{\gamma\lambda + \delta}(\gamma A + \delta I)x$$

Then if we apply the SDC algorithm to the matrix $(\gamma A + \delta I)^{-1}(\alpha A + \beta I)$, we can divide the spectrum with respect to a region

$$\Re\left(\frac{\alpha\lambda + \beta}{\gamma\lambda + \delta}\right) > 0.$$

For example, by computing the matrix sign function of $(A+(r-\mu)I)^{-1}(-A+(r+\mu)I)$, then the SDC algorithm will divide the spectrum of $A$ along a circle centered at $\mu$ with radius $r$. If $A$ is real, and we choose $\mu$ to be real, then all arithmetic will be real. Other more general regions can be obtained by taking $A_0$ as a polynomial function of $A$. For example, by computing the matrix sign function of $(A - \alpha I)^2$, we can divide the spectrum within a "bowtie" shaped region centered at $\alpha$. Figure 1 illustrates the regions which the algorithms can deal with assuming that $A$ is real and the algorithms use only real arithmetic.

# 3 Implementation and Performance

In this section, we report the implementation and performance the SDC algorithm with Newton iteration on distributed memory parallel machines, namely the Intel Delta and the CM-5. Implementations on workstations and shared memory machines can be found in [3].

## 3.1 Implementation and Performance on Intel Touchstone Delta

The Intel Touchstone Delta system, located at the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium, is $16 \times 32$ mesh of i860 processors with a wormhole routing interconnection network [27]. The communication characteristics are described in [29].

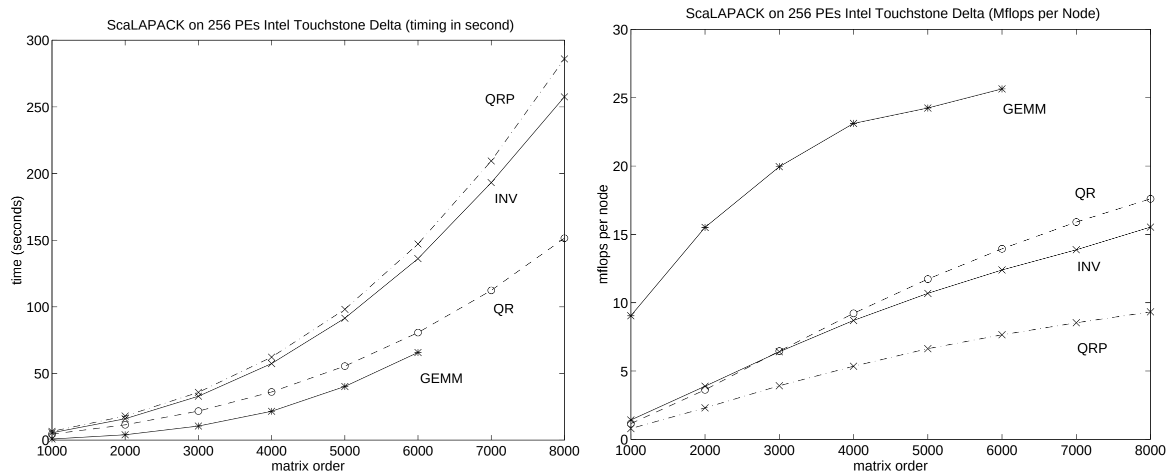


Figure 2: Performance of ScaLAPACK 1.0 subroutines on 256 ($16\times16$) PEs Intel Touchstone Delta system.

Our implementation on Intel is built upon ScaLAPACK 1.0 (beta version) [11]. A square block cyclic data decomposition scheme is used, which allows the routines to achieve well balanced computations and to minimize communication costs. ScaLAPACK relies on the Parallel Block BLAS (PB-BLAS) [12], which hides much of the interprocessor communication and makes it possible to avoid explicit calls to communication routines. The PB-BLAS itself is implemented on top of calls to the BLAS and to the Basic Linear Algebra Communication Subroutines (BLACS) [19].

The PUMMA routines [13] provide the required matrix multiplication. The matrix inversion is done in two steps. After the LU factorization has been computed, the upper triangular $U$ matrix is inverted, and $A^{-1}$ is obtained by substitution with $L$. Using blocked operations leads to performance comparable to that obtained for LU factorization. The implementation of the QR factorization with or without column pivoting is based on the parallel algorithm presented by Coleman and Plassmann [14]. The QR factorization with column pivoting has a much larger sequential component, processing one column at a time, and needs to update the norms of the column vectors at each step. This makes using blocked operations impossible and induces high synchronization overheads. However, as we will see, the cost of this step remains negligible in comparison with the time spent in the Newton iteration. The QR factorization without pivoting and the post- and pre-multiplication by an orthogonal matrix do use blocked operations. Two plots in Figure 2 are the timing and megaflops for the PUMMA package using the BLACS for matrix multiplication, and ScaLAPACK subroutines for the matrix inversion, QR decomposition with and without column pivoting.

To measure the efficiency of the algorithm for computing the spectral decomposition with respect to the pure imaginary axis, we generated random matrices with normal distribution (0,1). All computations were performed in real double precision arithmetic. Table 1 lists the CPU time and different megaflops rates. All the backward errors measured in $\|E_{21}\|_1/\|A\|_1$ are on the order of $10^{-12}$ to $10^{-13}$. It took between 18 to 21 steps of Newton

Table 1: The SDC algorithm with Newton iteration on 256-node Intel Touchstone Delta system.

| $n$ | Timing (seconds) | Mflops (total) | Mflops (per node) | GEMM-Mflops (per node) | INV-Mflops (per node) |
|---|---|---|---|---|---|
| 1000 | 134.22 | 293.05 | 1.14 | 9.04 | 1.41 |
| 2000 | 448.69 | 808.28 | 3.16 | 15.51 | 3.88 |
| 3000 | 792.18 | 1340.60 | 5.23 | 19.95 | 6.43 |
| 4000 | 1436.14 | 1841.98 | 7.19 | 23.12 | 8.70 |

Table 2: Performance Profile on 256-node Intel Touchstone Delta system.

| $n$ | Newton (%) | QRP(%) | $Q^TAQ$(%) | Total CPU |
|---|---|---|---|---|
| 1000 | 123.06(91%) | 6.87(5%) | 4.27(5%) | 134.22 |
| 2000 | 413.95(92%) | 18.60(4%) | 16.13(4%) | 448.69 |
| 3000 | 717.04(90%) | 36.76(5%) | 38.37(5%) | 792.18 |
| 4000 | 1300.16(90%) | 63.13(5%) | 72.80(5%) | 1436.14 |

iteration to converge. From Table 1, we see that for matrices of order 4000, the algorithm reached $7.19/23.12 \approx 31\%$ efficiency with respect to PUMMA matrix multiplication, and $7.19/8.70 \approx 82\%$ efficiency with respect to the underlying ScaLAPACK matrix inversion subroutine. Table 2 is the profile of the CPU time. It is clear that the Newton iteration (i.e., computing the matrix sign function) is most expensive, and takes about 90% of the total running time. Figure 3 shows the performance of the algorithm as a function of matrix size for different numbers of processors.

We also ran LAPACK driver routine DGEES (the standard serial QR algorithm) for computing the Schur decomposition on one i860 processor. It took 592 seconds for a matrix of order 600, or 9.1 megaflops. Assuming that the time scales like $n^3$, one can predict that for a matrix of order 4000, if the matrix was able to fit on a single node, then DGEES would take about 48 hours to compute the desired spectral decomposition. In contrast, the SDC algorithm would only take about 24 minutes. This is about 120 times faster. Of course, we should note that DGEES actually computes a complete Schur decomposition with the necessary reordering of the spectrum. Our algorithm only decomposes the spectrum along the pure imaginary axis. In some applications, this may be what users want. If the decomposition along a finer region or a complete Schur decomposition is desired, then the cost of the SDC algorithm will be increased, though it is likely that the first dividing step will take most of the time [9].

### 3.2 Implementation and Performance on the CM-5

Our implementation on 32 node CM-5 was carried out at the University of California at Berkeley. Each CM-5 node contains a 33 MHz Sparc with an FPU and 64 KB cache, four

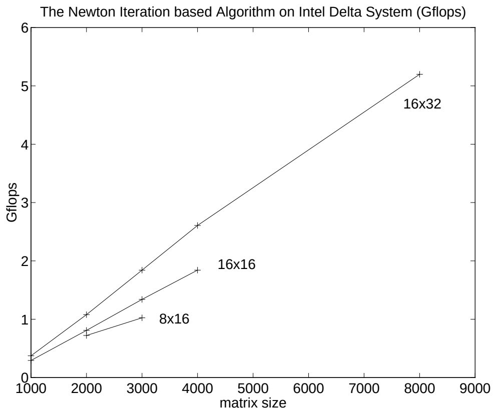


Figure 3: Performance of the SDC algorithm with Newton iteration on the Intel Delta system as a function of matrix size for different numbers of processors.

vector floating points units, and 32 MB of memory. The front end is a 33 HMz Sparc with 32 MB of memory. With the vector units, the peak 64-bit floating point performance is 128 megaflops per node (32 megaflops per vector unit). See [34] for more details.

The SDC algorithm is implemented in CM Fortran (CMF) version 2.1 – an implementation of Fortran 77 supplemented with array-processing extensions from the ANSI and ISO standard Fortran 90. CMF arrays come in two flavors. They can be distributed across CM processor memory (in some user defined layout) or allocated in normal column major fashion on the front end alone. When the front end computer executes a CM Fortran program, it performs serial operations on scalar data stored in its own memory, but sends any instructions for array operations to the nodes. On receiving an instruction, each node executes it on its own data. When necessary, CM nodes can access each other's memory by available communication mechanisms.

CMSSL (version 3.2) was used in our implementation. CMSSL provides data parallel implementations of many standard linear algebra routines. Figure 4 summarizes the performance of CMSSL routines underlying the implementation of the SDC algorithm. Matrix inversion is performed by solving the system $AX = I$. The LU factors can be obtained separately – to support Balzer's and Byers' scaling schemes to accelerate the convergence of Newton iteration – and there is a routine for estimating $\|A^{-1}\|_\infty$ from the LU factors to detect ill-conditioned intermediate matrices in the Newton iteration. The QR factorization with or without pivoting uses standard Householder transformations. Column blocking can be performed at the user's discretion to improve load balance and increase parallelism. The QR with pivoting routine is about half as fast as QR without pivoting. This is due in part to the elimination of blocking techniques when pivoting, as columns must be processed sequentially.

We tested the SDC algorithm with hybrid Newton-Schulz iteration for computing the spectral decomposition along the pure imaginary axis. The entries of random test matrices were uniformly distributed on $[-1, 1]$. We use the inequality $\|A_{i+1} - A_i\|_1 \leq \sqrt{n}$ as switching criterion from the Newton iteration (5) to the Newton-Schulz iteration (6), i.e., we relaxed

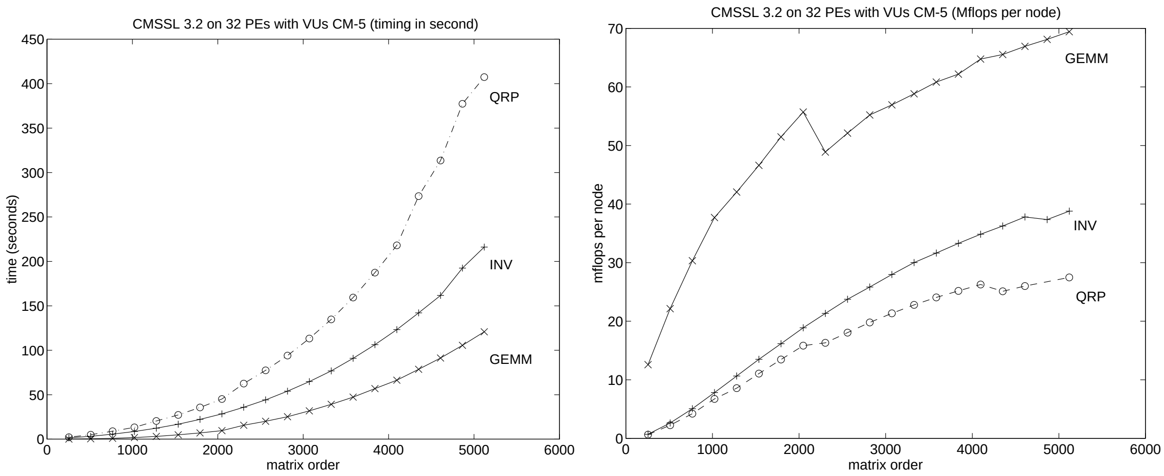


Figure 4: Performance of CMSSL 3.2 subroutines on 32 node CM-5 with vector units

Table 3: Performance of the SDC algorithm with Newton-Schultz iteration on a 32 node CM-5 with vector units.

| $n$ | Actual Time (seconds) | Predicted Time (seconds) | Mflops (total) | Mflops (per node) | GEMM-Mflops (per node) | Inverse-Mflops (per node) |
|---|---|---|---|---|---|---|
| 256 | 25 | 16 | 30.72 | 0.96 | 12.57 | 0.69 |
| 512 | 58 | 45 | 106.88 | 3.34 | 22.14 | 2.62 |
| 768 | 99 | 88 | 203.84 | 6.37 | 30.32 | 5.05 |
| 1024 | 143 | 146 | 318.40 | 9.95 | 37.71 | 7.81 |
| 1280 | 231 | 222 | 405.44 | 12.67 | 42.06 | 10.64 |
| 1536 | 296 | 316 | 520.64 | 16.27 | 46.61 | 13.49 |
| 1792 | 423 | 430 | 579.84 | 18.12 | 51.47 | 16.16 |
| 2048 | 506 | 567 | 732.16 | 22.88 | 55.72 | 18.87 |

the convergence condition $\|A_i^2 - I\| < 1$ for the Newton-Schulz iteration to

$$\|A_i^2 - I\|_1 = \|A_i(A_i - A_i^{-1})\|_1 = 2\|A_i(A_{i+1} - A_i)\|_1 \le 2\sqrt{n}\|A_i\|_1,$$

because this optimized performance over the test cases we ran.

Table 3 shows the CPU time in seconds and different megaflops rates. All backward errors are on the order of $10^{-13}$. The Newton iteration took between 14 to 16 steps to converge, and then took 2 steps of Newton-Schultz iteration. From the table, we see that by comparing to CMSSL matrix multiplication, we obtain 32% to 45% efficiency with the matrices sizes from 512 to 2048, even faster than the CMSSL matrix inverse subroutine. After profiling the total CPU time, we found that about 83% of total time is spent on the Newton iteration, 9% on the QR decomposition with pivoting, and 7.5% on the matrix multiplication for the Newton-Schulz iteration and orthogonal transformations.

# 4 Execution time modeling

In this section, we derive an execution time model for the SDC algorithm. We will show that the model confirms that the SDC algorithm scales well. The ratio of computation to communication, the chief determinant of scalability, is comparable to that required by current implementations of LU decomposition [11]. Since LU decomposition scales well to a wide variety of machines [17], the SDC algorithm can also be expected to scale well.

## 4.1 Details of the execution time model

Our model is based on the actual operation counts of the ScaLAPACK implementation and the following problem parameters and (measured) machine parameters:

| | |
|---|---|
| $n$ | Matrix size, |
| $p$ | Number of processors, |
| $\alpha$ | Time required to send a zero length message from one processor to another, |
| $\beta$ | Time required to send one double word from one processor to another, |
| $\gamma$ | Time required per BLAS3 floating point operation. |

The models for each of the building blocks, shown in Table 4, were created by counting the actual operations in the critical path. Each of these building block models were validated against the performance data shown in Figures 2 and 4.

In Table 5, the predicted running time of each of the steps of the algorithm is displayed. Summing the times in Table 5 yields:

$$\text{Total time} = 45\frac{n^3}{p}\gamma + (160 + 23\lg p)n\alpha + (90 + 40\lg p)\frac{n^2}{\sqrt{p}}\beta. \qquad (7)$$

Using the measured machine parameters given in Table 6 with equation (7) yields the predicted times on CM-5 (Table 3) and the Intel Delta system (Table 7). As Table 7 shows, our model underestimates the actual time on the Delta by no more than 30% for the problem and machine sizes listed. Table 3 shows that our model matches the performance on the CM-5 to within 25% for all problem sizes except the smallest, i.e. $n = 256$.

Table 4: Models for each of the building blocks

| Task | Computation Cost | Communication Cost latency | bandwidth$^{-1}$ |
|---|---|---|---|
| LU | $\frac{2}{3}\frac{n^3}{p}\gamma$ | $(6+\lg p)n\alpha$ | $(3+\frac{\lg p}{4})\frac{n^2}{\sqrt{p}}\beta$ |
| TRI | $\frac{4}{3}\frac{n^3}{p}\gamma$ | $2n\alpha$ | $(2+\frac{3}{2}\lg p)\frac{n^2}{\sqrt{p}}\beta$ |
| Matrix multiply | $2\frac{n^3}{p}\gamma$ | $(1+\frac{\lg p}{2})\sqrt{p}\alpha$ | $(1+\frac{\lg p}{2})\frac{n^2}{\sqrt{p}}\beta$ |
| QR | $\frac{4}{3}\frac{n^3}{p}\gamma$ | $3n\lg p\alpha$ | $\frac{3\lg p}{4}\frac{n^2}{\sqrt{p}}\beta$ |
| Householder application | $2\frac{n^3}{p}\gamma$ | | $2\frac{n^2}{\sqrt{p}}\lg p\beta$ |

Table 5: Model of the SDC algorithm with Newton iteration

| | | 20 matrix inversions | QR | 2 Householder applications | Total |
|---|---|---|---|---|---|
| Computation cost | $\times\frac{n^3}{p}\gamma$ | 40 | $\frac{4}{3}$ | 4 | 45 |
| Latency cost | $\times n\alpha$ | $160+20\lg p$ | $3\lg p$ | | $160+23\lg p$ |
| Bandwidth cost | $\times\frac{n^2}{\sqrt{p}}\beta$ | $90+35\lg p$ | $\frac{3}{4}\lg p$ | 4 | $90+40\lg p$ |

Table 6: Machine parameters

| Model Parameter | Description | Performance limited by | measured values $\mu$s CM-5 | Delta |
|---|---|---|---|---|
| $\gamma$ | time per BLAS 3 flop | peak flop rate | 1/90 | 1/34 |
| $\alpha$ | message latency | comm. software | 150 | 157 |
| $\beta$ | bandwidth$^{-1}$ | comm. hardware | 1.62 | 1.67 |

Table 7: Actual and predicted performance of the SDC algorithm with Newton iteration for the spectral decomposition along the pure imaginary axis

| **Delta** | $8 \times 16$ PEs | | $16 \times 16$ PEs | | $16 \times 32$ PEs | |
|---|---|---|---|---|---|---|
| $n$ | actual time (sec) | predicted time (sec) | actual time (sec) | predicted time (sec) | actual time (sec) | predicted time ( sec) |
| 1000 | – | – | 134 | 102 | 110 | 93 |
| 2000 | 502 | 402 | 448 | 320 | 336 | 269 |
| 3000 | 1037 | 921 | 792 | 687 | 576 | 542 |
| 4000 | – | – | 1436 | 1231 | 1014 | 927 |
| 8000 | – | – | – | – | 4268 | 3910 |

Table 8 compares the execution time cost to divide the spectrum once by the SDC algorithm with the cost for LU decomposition. The ratio of SDC to LU costs in each of the three categories, the cost of a flop, message initiation cost and inverse bandwidth cost, is shown in the third column and also displayed here:

$$\left\langle 67, \frac{160 + 23 \lg p}{6 + \lg p}, \frac{90 + 40 \lg p}{3 + \frac{1}{4} \lg p} \right\rangle$$

These cost ratios vary slowly with the number of processors. For example, the cost of splitting the spectrum once with the SDC algorithm on 1000 processors is 67 times the flop cost in LU, 24 times the message initiation cost in LU and 90 times the inverse bandwidth cost in LU. At the extremes, $p = 1$ and $p = \infty$, the cost raios are $< 67, 27, 30 >$, and $< 67, 23, 160 >$ respectively. These cost ratios show that the SDC algorithm will scale almost as well as the LU decomposition across most computers.

The performance figures in Table 6 are all measured by an independent program, except for the CM-5 Level 3 BLAS performance. The communication performance figures for the Delta in Table 6 are from a report by Littlefield[1] [29]. The communication performance figures for the CM-5 are as measured by Whaley[2] [35]. The computation performance for the Delta is from the LINPACK benchmark[17] for a one processor Delta. There is no entry for a one processor CM-5 in the LINPACK benchmark, so $\gamma$ for the CM-5 in Table 6 is chosen from our own experience.

## 4.2 Discrepancies between the model and actual times

There are numerous sources of error in our model. The model does not count all floating point operations. The number of processor rows and columns and the block size affect performance and ignoring them therefore contributes to the total error. Communications do not exactly fit the linear model $(\alpha + n\beta)$, nor are matrix multiply costs constant per flop.

[1]The BLACS use protocol 2, and the communication pattern most closely resembles the "shift" timings.
[2]$\alpha$ is from Table 8 and $\beta$ is from Table 5 in [35].

Table 8: The scalability of the SDC algorithm versus LU decomposition

| | SDC | LU | SDC/ LU |
|---|---|---|---|
| Computation Cost | $45\frac{n^3}{p}\gamma$ | $\frac{2}{3}\frac{n^3}{p}\gamma$ | 67 |
| Latency cost | $(160+23\lg p)n\alpha$ | $(6+\lg p)n\alpha$ | $\frac{160+23lg(p)}{6+lg(p)}$ |
| Bandwidth cost | $(90+40\lg p)\frac{n^2}{\sqrt{p}}\beta$ | $(3+\frac{\lg p}{4})\frac{n^2}{\sqrt{p}}\beta$ | $\frac{90+40lg(p)}{3+\frac{1}{4}lg(p)}$ |

The model assumes that exactly 20 Newton iterations are required, wheras the actual number varied from 18 to 22. It is based on QR decomposition without pivoting, but the code has to use the QR decomposition with pivoting. On the CM-5, timings are from the Newton-Shultz iteration because that the latter is slightly more efficient, but the model is uniformally based on the Newton iteration.

We have a more detailed model which matches the performance better than the one shown here. The detailed model shows that algorithmic discrepancies and load imbalance contribute the largest errors for large problems ($n > 512$) while uncounted operations contribute the largest error for small problems ($n < 512$).

## 5 XI : A Graphical User Interface to SDC

To take advantage of the graphical nature of the spectral decomposition process of the SDC algorithm, a graphical user interface has been implemented. Written in C and based on X11R5's standard Xlib library, the Xt toolkit and MIT's Athena widget set, it has been nicknamed **XI** for "X11 Interface". The programmer's interface to **XI** consists of seven subroutines designed independently of any specific SDC implementation. Thus **XI** can be attached to any SDC code. At present, it is in use with the CM-5 CMF implementation and Fortran 77 version of the SDC algorithm. Figure 5 shows the coupling of the SDC code and the **XI** library of subroutines.

Basically, the SDC code calls an **XI** routine which handles all interaction with the user and returns only when it has the next request for a parallel computation. The SDC code processes this request on the parallel engine, and if necessary calls another **XI** routine to inform the user of the computational results. If the user had selected to split the spectrum, then at this point the size of the highlighted region, and the error bound on the computation (along with some performance information) is reported, and the user is given the choice of confirming or refusing the split. Appropriate action is taken depending on the choice. This process is repeated until the user decides to terminate the program.

All data structures pertaining to the matrix decomposition process are managed by **XI**. A binary tree records the size and status (solved/not solved) of each diagonal block corresponding to a spectral region, the error bounds of each split, and other information. Having the X11 interface manage the decomposition data frees the programmer of these responsibilities and encapsulates the decomposition process. The programmer obtains any useful information via the interface subroutines.

Figure 6 pictures a sample session of the user interface on the CM-5 with a matrix of

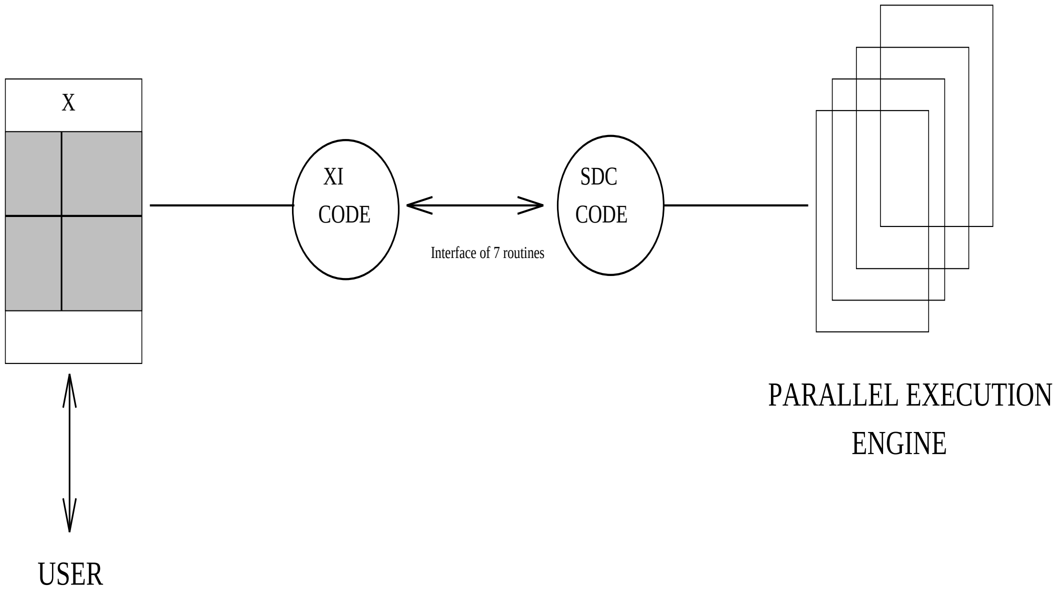


Figure 5: The X11 Interface (**XI**) and SDC

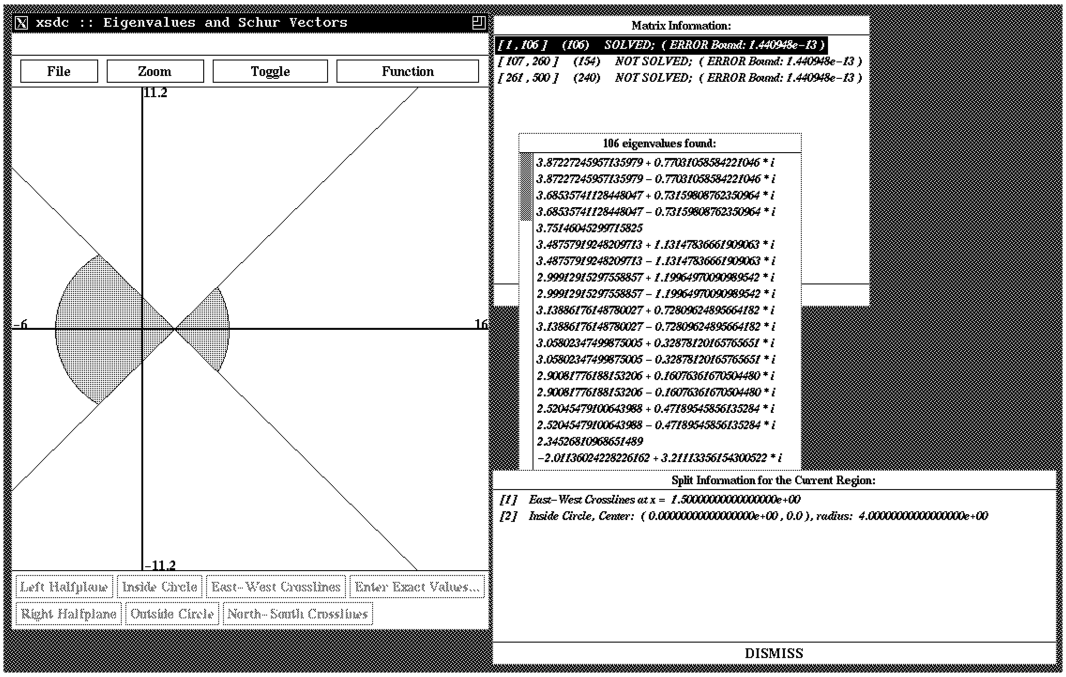


Figure 6: A sample *xsdc* session

order 500. The central window (called the "spectrum window") represents the region of the complex plane indicated by the axes. Its title – "xsdc :: Eigenvalues and Schur Vectors" – indicates that the task is to compute eigenvalues and Schur vectors for the underline matrix. The lines on the spectrum window (other than the axes) are the result of spectral divide-and-conquer, while the shading indicates that the "bowtie" region of the complex plane is currently selected for further analysis. The other windows show the details of the process.

The buttons at the top control I/O, the appearance of the spectrum window, and algorithmic choices:

- `File` lets one save the matrix, start on a new matrix, or quit.
- `Zoom` lets one navigate around the complex plane by zooming in or out on part of the spectrum window.
- `Toggle` turns on or off the features of the spectrum window (for example the axes, Gershgorin disks, eigenvalues).
- `Function` lets one modify the algorithm, or display details about the progress being made.

The buttons at the bottom are used in splitting the spectrum. For example clicking on `Right halfplane` and then clicking at any point on the spectrum window will split the spectrum into two halfplanes at that point, with the right halfplane selected for further division. The `Split Information` window keeps track of the matrix splitting process. The `Matrix Information` window displays the status of the matrix decomposition process, where each of the three entries corresponds to a spectral region and a square diagonal block of the $3 \times 3$ block upper triangular matrix, and informs us of the block's size, whether its eigenvalues (eigenvectors, Schur vectors) have been computed or not, and the maximum error bound encountered. The listed eigenvalues can be plotted on the spectrum at the user's request.

The user may select any region of the complex plane (and hence any sub-matrix on the diagonal) for further decomposition by clicking the pointer in the desired region. Once a block is small enough, the user may choose to solve it via the `Function` button at the top of the spectrum window.

# 6 Concluding Remarks

Our implementation of the SDC algorithm with Newton iteration uses only highly efficient matrix computation kernels, which are available in the public domain and from distributed memory parallel computer vendors. The performance attained is encouraging. This approach merits consideration for other numerical algorithms. The object oriented user interface XI provides a paradigm for use in the future to design a more user friendly interface in the massively parallel computing environment. We note that all the approaches discussed here can be extended to compute the both right and left deflating subspaces of a regular matrix pencil $A - \lambda B$.

As the spectrum is repeatedly partitioned in a divide-and-conquer fashion, there is obviously task parallelism available because of the independent submatrices that arise, as well as the data parallel-like matrix operations considered in this paper. Analysis in [9] indicates that this task parallelism can contribute at most a small constant factor speedup, since most of the work is at the root of the divide-and-conquer tree. This can simplify the implementation.

## Acknowledgements

Bai and Demmel were supported in part by an ARPA grant. Demmel and Petitet were supported in part by NSF grant ASC-9005933, Demmel, Dongarra and Robinson were supported in part by ARPA contact DAAL03-91-C-0047 administered by the Army Research Office. Dongarra was also supported in part by DOE under contract DE-AC05-84OR21400. Ken Stanley was supported by an NSF graduate student fellowship.

This work was performed in part using the Intel Touchstone Delta System operated by the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided through the Center for Research on Parallel Computing.

## References

[1] 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 (second edition).* SIAM, Philadelphia, 1995.

[2] L. Auslander and A. Tsao. On parallelizable eigensolvers. *Adv. in Appl. Math.*, 13:253–261, 1992.

[3] Z. Bai and J. Demmel. Design of a parallel nonsymmetric eigenroutine toolbox, Part I. In R. F. Sincovec *et al*, editor, *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing.* SIAM, 1993.

[4] Z. Bai, J. Demmel, and M. Gu. Inverse free parallel spectral divide and conquer algorithms for nonsymmetric eigenproblems. Department of Mathematics RR 94-01, University of Kentucky, 1994. to appear *in Numer. Math.*

[5] A. N. Beavers and E. D. Denman. A computational method for eigenvalue and eigenvectors of a matrix with real eigenvalues. *Numer. Math.*, 21:389–396, 1973.

[6] C. Bischof and X. Sun. A divide and conquer method for tridiagonalizing symmetric matrices with repeated eigenvalues. Technical Report, Argonne National Lab, Argonne, IL.

[7] R. Byers. Solving the algebraic Riccati equation with the matrix sign function. *Lin. Alg. Appl.*, 85:267–279, 1987.

[8] R. Byers, C. He, and V. Mehrmann. The matrix sign function method and the computation of invariant subspaces. Technical Report Preprint SPC 94-25, Fakultät für Mathematik, TU Chemnitz-Zwickau, Germany, 1994.

[9] S. Chakrabarti, J. Demmel, and K. Yelick. On the benefit of mixed parallelism. Computer Science Division Technical Report, University of California, Berkeley, 1994. submitted to IPPS.

[10] T. Chan. Rank revealing QR factorizations. *Lin. Alg. Appl.*, 88/89:67–82, 1987.

[11] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. Computer Science Dept. Technical Report CS-95-283, University of Tennessee, Knoxville, 1995. (LAPACK Working Note #95).

[12] J. Choi, J. Dongarra, and D. Walker. PB-BLAS: A set of paralle block basic linear algebra subprograms. Technical Report, University of Tennessee, Knoxville, 1993. available in postscript from netlib/scalapack.

[13] J. Choi, J. Dongarra, and D. Walker. PUMMA: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers. Computer Science Dept. Technical Report CS-93-187, University of Tennessee, Knoxville, 1993. (LAPACK Working Note #57).

[14] T. Coleman and P. Plassmann. A parallel nonlinear least-squares solver: Theoretical analysis and numerical results. *SIAM J. Sci. Comput.*, 13(3):771–793, 1992.

[15] J. Demmel. Trading off parallelism and numerical stability. In G. Golub M. Moonen and B. de Moor, editors, *Linear Algebra for Large Scale and Real-Time Applications*, pages 49–68. Kluwer Academic Publishers, 1993.

[16] 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.

[17] J. Dongarra. Performance of various computers using standard linear equations software. Computer science dept. technical report, University of Tennessee, Knoxville, TN, January 1993.

[18] J. Dongarra and M. Sidani. A parallel algorithm for the non-symmetric eigenvalue problem. *SIAM J. Sci. Comp.*, 14(3):542–569, May 1993.

[19] J. Dongarra, R. van de Geijn, and R. Whaley. A Users' Guide to the BLACS. Technical Report, University of Tennessee, Knoxville, 1993. available in postscript from netlib/scalapack.

[20] M. Gu and S. Eisenstat. An efficient algorithm for computing a rank-revealing QR decomposition. Computer Science Dept. Report YALEU/DCS/RR-916, Yale University, 1993.

[21] G. Henry and R. van de Geijn. Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problems: myths and reality. CS-TR-94-244, University of Tennessee, Knoxville, August 1994. (LAPACK Working Note #79).

[22] N. J. Higham. Computing the polar decomposition - with applications. *SIAM J. Sci. Stat. Comput.*, 7:1160–1174, 1986.

[23] J. Howland. The sign matrix and the separation of matrix eigenvalues. *Lin. Alg. Appl.*, 49:221–232, 1983.

[24] S. Huss-Lederman, A. Tsao, and G. Zhang. A parallel implementation of the invariant subspace decomposition algorithm for dense symmetric matrices. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1993.

[25] C. Kenney and A. Laub. Rational iteration methods for the matrix sign function. *SIAM J. Mat. Anal. Appl.*, 21:487–494, 1991.

[26] T.-Y. Li and Z. Zeng. Homotopy-determinant algorithm for solving nonsymmetric eigenvalue problems. *Math. Comp.*, 59(200):483–502, October 1992.

[27] S. L. Lillevik. The Touchstone 30 Gigaflop DELTA prototype. In *Sixth Distributed Memory Computing Conference Proceedings*. IEEE Computer Society Press, 1991.

[28] 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.

[29] A. Littlefield. Characterizing and tuning communication performance on the Touchstone DELTA and iPSC/860. In *Proceedings of the 1992 Intel User's Group Meeting*, Dallas, TX, October 4-7 1992.

[30] A. N. Malyshev. Parallel algorithm for solving some spectral problems of linear algebra. *Lin. Alg. Appl.*, 188,189:489–520, 1993.

[31] J. Roberts. Linear model reduction and solution of the algebraic Riccati equation. *Inter. J. Control*, 32:677–687, 1980.

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

[33] 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.

[34] Thinking Machine Corporation, Cambridge, Massachusetts. *Connection Machine CM-5 Technical Summary*, 1992.

[35] R. Whaley. Basic linear algebra communication subroutines: analysis and implementation across multiple parallel architecture. CS-TR-94-234, University of Tennessee, Knoxville, May 1994. (LAPACK Working Note #73).