The Spectral Decomposition of Nonsymmetric Matrices on Distributed Memory Parallel Computers

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

January 9, 1995

Abstract

We study the implementation and performance of a class of algorithms for finding eigenvalues of nonsymmetric matrices on distributed memory parallel computers. The algorithms perform *spectral divide and conquer*, i.e. they recursively divide the matrix into smaller submatrices, each of which has a subset of the original eigenvalues as its own. One algorithm uses the matrix sign function evaluated with Newton iteration. The other algorithm avoids the matrix inverse required by Newton iteration, and so is called the inverse free algorithm. Both algorithms are simply constructed from a small set of highly parallelizable matrix building blocks, including matrix multiplication, QR decomposition and matrix inversion. Efficient implementations of these building blocks are available on many machines. The price paid for easy parallelization of these algorithms is potential loss of stability compared to the best serial algorithm in some ill-conditioned cases. Fortunately, the program can detect and compensate for this loss of stability.

In our implementation of the sign function algorithm on a 256 processor Intel Touchstone Delta system, the algorithm reached 31% efficiency with respect to the underlying PUMMA matrix multiplication on 4000-by-4000 matrices, and 82% efficiency with respect to the underlying ScaLAPACK 1.0 (beta version) matrix inversion. On a 32 node Thinking Machines CM-5 with vector units, on 2048-by-2048 matrices the algorithm reached 41% efficiency with respect to the matrix multiplication in CMSSL 3.2. Our performance model predicts the performance reasonably accurately.

To take advantage of the geometric nature of the spectral decomposition algorithm, we have also designed a graphical user interface to let the user choose which eigenvalues to compute.

1 Introduction

A standard technique in parallel computing is to build new algorithms from existing high performance building blocks. For example, the LAPACK linear algebra library [1] is writ-

*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.

ten in terms of the Basic Linear Algebra Subroutines (BLAS)[38, 23, 22], for which efficient implementations are available on many workstations, vector processors, and shared memory parallel machines. The recently released ScaLAPACK 1.0(beta) linear algebra library [26] is written in terms of the Parallel Block BLAS (PB-BLAS) [15], Basic Linear Algebra Communication Subroutines (BLACS) [25], 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)[54] provides analogous functionality and high performance for the CM-5.

In this work, we use these high performance kernels to build two new algorithms for finding eigenvalues and invariant subspaces of nonsymmetric matrices on distributed memory parallel computers. These algorithms perform *spectral divide and conquer*, i.e. they recursively divide the matrix into smaller submatrices, each of which has a subset of the original eigenvalues as its own. One algorithm uses the matrix sign function evaluated with Newton iteration [8, 42, 6, 4]. The other algorithm avoids the matrix inverse required by Newton iteration, and so is called the inverse free algorithm [30, 10, 44, 7]. Both algorithms are simply constructed from a small set of highly parallelizable building blocks, including matrix multiplication, QR decomposition and matrix inversion, as we describe in section 2.

By using existing high performance kernels in ScaLAPACK and CMSSL, we have achieved high efficiency. On a 256 processor Intel Touchstone Delta system, the sign function algorithm reached 31% efficiency with respect to the underlying matrix multiplication (PUMMA [16]) for 4000-by-4000 matrices, and 82% efficiency with respect to the underlying ScaLAPACK 1.0 matrix inversion. On a 32 processor Thinking Machines CM-5 with vector units, the hybrid Newton-Schultz sign function algorithm obtained 41% efficiency with respect to matrix multiplication from CMSSL 3.2 for 2048-by-2048 matrices.

The nonsymmetric spectral decomposition problem has until recently resisted attempts at parallelization. The conventional method is to use the Hessenberg QR algorithm. One first reduces the matrix to Schur form, and then swaps the desired eigenvalues along the diagonal to group them together in order to form the desired invariant subspace [1]. The algorithm had appeared to required fine grain parallelism and be difficult to parallelize [5, 27, 57], but recently Henry and van de Geijn[32] have shown that the Hessenburg QR algorithm phase can be effectively parallelized for distributed memory parallel computers with up to 100 processors. Although parallel QR does not appear to be as scalable as the algorithms presented in this paper, it may be faster on a wide range of distributed memory parallel computers. Our algorithms perform several times as many floating point operations as QR, but they are nearly all within Level 3 BLAS, whereas implementations of QR 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.

Other parallel eigenproblem algorithms which have been developed include earlier parallelizations of the QR algorithm [29, 50, 56, 55], Hessenberg divide and conquer algorithm using either Newton's method [24] or homotopies [17, 39, 40], and Jacobi's method [28, 47, 48, 49]. 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 [20], or all three.

The methods in this paper may be less stable than QR algorithm, and may fail to converge in a number of circumstances. Fortunately, it is easy to detect and compensate for this loss of stability, by choosing to divide the spectrum in a slightly different location. Compared with the other approaches mentioned above, we believe the algorithms discussed in this paper offer an effective tradeoff between parallelizability and stability.

The other algorithms most closely related to the approaches used here may be found in [3, 9, 36], where symmetric matrices, or more generally matrices with real spectra, are treated.

Another advantage of the algorithms described in this paper is that they can compute 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 will describe a graphical user interface for the algorithms.

The rest of this paper is organized as follows. In section 2, we present our two algorithms for spectral divide and conquer in a single framework, show how to divide the spectrum along arbitrary circles and lines in the complex plane, and discuss implementation details. In section 3, we discuss the performance of our algorithms on the Intel Delta and CM-5. In section 4, we present a model for the performance of our algorithms, and demonstrate that it can predict the execution time reasonably accurately. Section 5 describes the design of an X-window user interface. Section 6 draws conclusions and outlines our future work.

2 Parallel Spectral Divide and Conquer Algorithms

Both spectral divide and conquer (SDC) algorithms discussed in this paper can be presented in the following framework. Let

$$A = X \begin{pmatrix} J_+ & 0 \\ 0 & J_- \end{pmatrix} X^{-1} \tag{2.1}$$

be the Jordan canonical form of A, where the eigenvalues of the $l \times l$ submatrix J_+ are the eigenvalues of A inside a selected region $\mathcal{D}$ in the complex plane, and the eigenvalues of the $(n-l) \times (n-l)$ submatrix 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. The matrix

$$P_+ = X \begin{pmatrix} I & 0 \\ 0 & 0 \end{pmatrix} X^{-1} \tag{2.2}$$

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 Π 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:

$$Q^H A Q = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \tag{2.3}$$

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 (2.2), A_{11} will have the eigenvalues outside $\mathcal{D}$ and A_{22} will have the eigenvalues inside $\mathcal{D}$.

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

2.1 The SDC algorithm with Newton iteration

The first SDC algorithm uses the matrix sign function, which was introduced by Roberts [46] for solving the algebraic Riccati equation. However, it was soon extended to solving the spectral decomposition problem [8]. More recent studies may be found in [11, 42, 6].

The matrix sign function, $\text{sign}(A)$, of a matrix A with no eigenvalues on the imaginary axis can be defined via the Jordan canonical form of A (2.1), 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 $\bar{\mathcal{D}}$. 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

$$P_+ = \frac{1}{2}(I + \text{sign}(A)) \tag{2.4}$$

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 $\bar{\mathcal{D}}$.

Now let $P_+ = QR\Pi$ be the rank revealing QR decomposition of the projector P_+. Then Q yields the desired spectral decomposition (2.3), 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 $\bar{\mathcal{D}}$.

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

$$A_{j+1} = \frac{1}{2}(A_j + A_j^{-1}), \quad j = 0, 1, 2, \ldots \quad \text{with} \quad A_0 = A. \tag{2.5}$$

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 [46, 35]. The iteration fails otherwise, and in finite precision, 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 [12, 33, 37]. For example, if $\|A^2 - I\| < 1$, we may use the so called Newton-Schulz iteration

$$A_{i+1} = \frac{1}{2}A_i(3I - A_i^2) \quad \text{with} \quad A_0 = A \tag{2.6}$$

to avoid the use of the matrix inverse. Although it requires twice as many flops, 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 (we discuss the performance of one such hybrid later).

Hence, we have the following algorithm which divides the spectrum along the pure imaginary axis.

ALGORITHM 1 (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 $\|A_{j+1} - A_j\|_1 \le \tau \|A_j\|_1$, $p = j + 1$, exit
End for;
Compute $\frac{1}{2}(A_p + I) = QR\Pi$; (rank revealing QR decomposition)
Let $l = \text{rank}(R)$;
Compute $Q^H A Q = \begin{pmatrix} A_{11} & A_{12} \\ E_{21} & A_{22} \end{pmatrix}$;
Compute $\|E_{21}\|_1 / \|A\|_1$.

Here τ is the stopping criterion for the Newton iteration (say, $\tau = n\varepsilon$, where ε is the machine precision), and $j_{\max}$ limits the maximum number of iterations (say $j_{\max} = 60$). 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 just use the QR decomposition with column pivoting to reveal rank, although more sophisticated rank-revealing schemes exist [14, 31, 34, 51].

All the variations of the Newton iteration with global convergence still 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 is discussed in [11, 6, 4] and the references therein.

2.2 The SDC algorithm with inverse free iteration

The above algorithm needs an explicit matrix inverse. This could cause numerical instability when the matrix is ill-conditioned. The following algorithm, originally due to Godunov, Bulgakov and Malyshev [30, 10, 44] and modified by Bai, Demmel and Gu [7], eliminates the need for the matrix inverse, and divides the spectrum along the unit circle instead of the imaginary axis. We first describe the algorithm, and then briefly explain why it works.

Algorithm 2 (The SDC Algorithm with Inverse Free Iteration)

Let $A_0 = A$, $B_0 = I$;
For $j = 0, 1, \ldots$ until convergence or $j > j_{\max}$ do
$\begin{pmatrix} B_j \\ -A_j \end{pmatrix} = \begin{pmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{pmatrix} \begin{pmatrix} R_j \\ 0 \end{pmatrix}$; (QR decomposition)
$A_{j+1} = Q_{12}^H A_j$;
$B_{j+1} = Q_{22}^H B_j$;
if $(j > 1 \ \& \ \|R_j - R_{j-1}\|_1 \leq \tau \|R_{j-1}\|_1)$, $p = j + 1$, exit
End for;
Compute $(A_p + B_p)^{-1} B_p = QR\Pi$; (rank revealing QR decomposition)
Let $l = \mathrm{rank}(R)$;
Compute $Q^H A Q = \begin{pmatrix} A_{11} & A_{12} \\ E_{21} & A_{22} \end{pmatrix}$;
Compute $\|E_{21}\|_1 / \|A\|_1$.

As in Algorithm 1, we need to choose a stopping criterion τ in the inner loop, as well as a limit $j_{\max}$ on the maximum number of iterations. On convergence, the eigenvalues of A_{11} are the eigenvalues of A inside the unit disk $\mathcal{D}$, and the eigenvalues of A_{22} are the eigenvalues of A outside $\mathcal{D}$. It is assumed that no eigenvalues of A are on the unit circle. As with Algorithm 1, the quantity $\|E_{21}\|_1 / \|A\|_1$ measures the backward stability.

To illustrate how the algorithm works we will assume that all matrices we want to invert are invertible. From the inner loop of the algorithm, we see that

$$\begin{pmatrix} Q_{11}^H & Q_{21}^H \\ Q_{12}^H & Q_{22}^H \end{pmatrix} \begin{pmatrix} B_j \\ -A_j \end{pmatrix} = \begin{pmatrix} Q_{11}^H B_j - Q_{21}^H A_j \\ Q_{12}^H B_j - Q_{22}^H A_j \end{pmatrix} = \begin{pmatrix} R_j \\ 0 \end{pmatrix}$$

so $Q_{12}^H B_j = Q_{22}^H A_j$ or $B_j A_j^{-1} = Q_{12}^{-H} Q_{22}^H$. Therefore

$$A_{j+1}^{-1} B_{j+1} = A_j^{-1} Q_{12}^{-H} Q_{22}^H B_j = (A_j^{-1} B_j)^2$$

so the algorithm is simply repeatedly squaring the eigenvalues, driving the ones inside the unit circle to 0 and those outside to ∞. Repeated squaring yields quadratic convergence. This is analogous to the sign function iteration where computing $(A + A^{-1})/2$ is equivalent to taking the Cayley transform $(A - I)(A + I)^{-1}$ of A, squaring, and taking the inverse Cayley transform. Further explanation of how the algorithm works can be found in [7].

An attraction of this algorithm is that it can equally well deal with the generalized nonsymmetric eigenproblem $A - \lambda B$, provided the problem is regular, i.e. $\det(A - \lambda B)$ is not identically zero. One simply has to start the algorithm with $B_0 = B$ instead of $B_0 = I$.

Regarding the QR decomposition in the inner loop, there is no need to form the entire $2n \times 2n$ unitary matrix Q in order to get the submatrices Q_{12} and Q_{22}. Instead, we can compute the QR decomposition of the $2n \times n$ matrix $(B_j^H, -A_j^H)^H$, which leaves Q stored implicitly as Householder vectors in the lower triangular part of the matrix and another n

dimensional array. We can then apply Q — without computing it — to the $2n \times n$ matrix $(0, I)^T$ to obtain the desired matrices Q_{12} and Q_{22}.

We now show how to compute Q in the rank revealing QR decomposition of $(A_p + B_p)^{-1} A_p$ without computing the explicit inverse $(A_p + B_p)^{-1}$ and subsequent products. This will yield the ultimate *inverse free* algorithm. Recall that for our purposes, we only need the unitary factor Q and the rank of $(A_p + B_p)^{-1} A_p$. It turns out that by using the generalized QR decomposition technique developed in [45, 2], we can get the desired information without computing $(A_p + B_p)^{-1}$. In fact, in order to compute the QR decomposition with pivoting of $(A_p + B_p)^{-1} A_p$, we first compute the QR decomposition with pivoting of the matrix A_p:

$$A_p = Q_1 R_1 \Pi, \tag{2.7}$$

and then we compute the RQ factorization of the matrix $Q_1^H (A_p + B_p)$:

$$Q_1^H (A_p + B_p) = R_2 Q. \tag{2.8}$$

From (2.7) and (2.8), we have $(A_p + B_p)^{-1} A_p = Q^H (R_2^{-1} R_1) \Pi$. The Q is the desired unitary factor. The rank of R_1 is also the rank of the matrix $(A_p + B_p)^{-1} A_p$.

2.3 Spectral Transformation Techniques

Although Algorithms 1 and 2 only divide the spectrum along the pure imaginary axis and the unit circle, respectively, 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 a major attraction of this kind of algorithm.

Let us show how to use Möbius transformations to divide the spectrum along arbitrary lines and circles. Transform the eigenproblem $Az = \lambda z$ to

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

Then if we apply Algorithm 1 to $A_0 = (\gamma A + \delta I)^{-1} (\alpha A + \beta I)$ we can split the spectrum with respect to a region

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

If we apply Algorithm 2 to $(A_0, B_0) = (\alpha A + \beta I, \gamma A + \delta I)$, we can split along the curve

$$\left| \frac{\alpha \lambda + \beta}{\gamma \lambda + \delta} \right| = 1.$$

For example, by computing the matrix sign function of $(A + (r - \mu) I)^{-1} (-A + (r + \mu) I)$, then Algorithm 1 will split the spectrum of A along a circle centered at μ with radius r. If A is real, and we choose μ to be real, then all arithmetic will be real.

If $A_0 = A - \mu I$ and $B_0 = rI$, then Algorithm 2 will split the spectrum of A along a circle centered at μ with radius r. If A is real, and we choose μ to be real, then all arithmetic in the algorithm will be real.

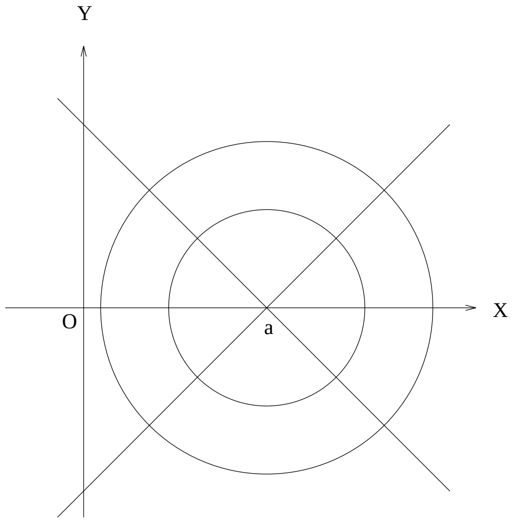

Figure 1: Different Geometric Regions for the Spectral Decomposition

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 α. Figure 1 illustrates the regions which the algorithms can deal with assuming that A is real and the algorithms use only real arithmetic.

2.4 Tradeoffs

Algorithm 1 computes an explicit inverse, which could cause numerical instability if the matrix is ill-conditioned. The inverse free Algorithm 2 provides an alternative approach for achieving better numerical stability. There are some very difficult problems where Algorithm 2 gives a more accurate answer than Algorithm 1. Numerical examples can be found in [7]. However, neither algorithm avoids all accuracy and convergence difficulties associated with eigenvalues very close to the boundary of the selected region.

The stability advantage of the inverse free approach is obtained at the cost of more storage and arithmetic. Algorithm 2 needs $4n^2$ more storage space than Algorithm 1. This will certainly limit the problem size we will be able to solve. Furthermore, one step of the Algorithm 2 does about 6 to 7 times more arithmetic than the one step of Algorithm 1. QR decomposition, the major component of Algorithm 2, and matrix inversion, the main component of Algorithm 1, require comparable amounts of communication per flop. (See table 4 for details.) Therefore, Algorithm 2 can be expected to run significantly slower than Algorithm 1.

Since Algorithm 1 is faster but somewhat less stable than Algorithm 2, and since testing stability is easy (compute $\|E_{21}\|_1/\|A\|_1$), we may use the following 3 step algorithm:

1. Try to use Algorithm 1 to split the spectrum. If it succeeds, stop.

2. Otherwise, try to split the spectrum using Algorithm 2. 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 [19].

If a fast parallel version of the QR algorithm[32] becomes available, it would probably be faster than the inverse free algorithm and hence would obviate the need for the second step listed above. Algorithm 2 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 $A - \lambda B$.

3 Implementation and Performance

We started with a Fortran 77 implementation of Algorithm 1. This code is built using the `BLAS` and `LAPACK` for the basic matrix operations, such as LU decomposition, triangular inversion, QR decomposition and so on. Initially, we tested our software on SUN and IBM RS6000 workstations, and then the CRAY. Some preliminary performance data of the matrix sign function based algorithm have been reported in [6]. In this report, we will focus on the implementation and performance evaluation of the algorithms on distributed memory parallel machines, namely the Intel Delta and the CM-5.

We have implemented Algorithm 1, and collected a large set of data for the performance of the primitive matrix operation subroutines on our target machines. More performance evaluation and comparison of these two algorithms and their applications are in progress.

3.1 Implementation and Performance on Intel Touchstone Delta

The Intel Touchstone Delta computer system is 16×32 mesh of i860 processors with a wormhole routing interconnection network [41], located at the California Institute of Technology on behalf of the Concurrent Supercomputing Consortium. The Delta's communication characteristics are described in [43].

In order to implement Algorithm 1, it was natural to rely on the ScaLAPACK 1.0 library (beta version) [26]. This choice requires us to exploit two key design features of this package. First, the ScaLAPACK library relies on the Parallel Block BLAS (`PB-BLAS`)[15], which hides much of the interprocessor communication. This hiding of communication makes it possible to express most algorithms using only the `PB-BLAS`, thus avoiding explicit calls to communication routines. The `PB-BLAS` are implemented on top of calls to the `BLAS` and to the Basic Linear Algebra Communication Subroutines (`BLACS`)[25]. Second, ScaLAPACK assumes that the data is distributed according to the square block cyclic decomposition scheme, which allows the routines to achieve well balanced computations and to minimize communication costs. ScaLAPACK includes subroutines for LU, QR and Cholesky factorizations, which we use as building blocks for our implementation. The PUMMA routines [16] provide the required matrix multiplication.

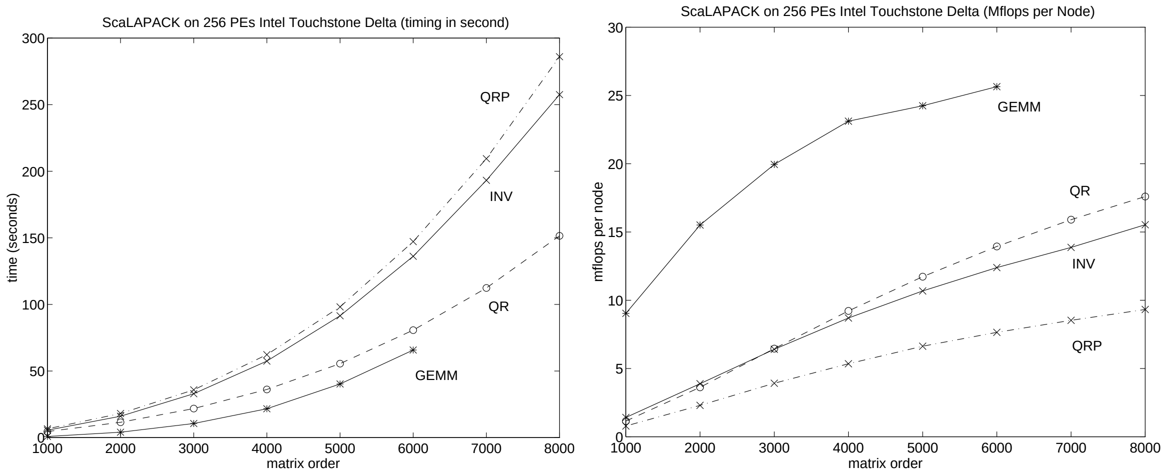

Figure 2: Performance of ScaLAPACK 1.0 (beta version) subroutines on 256 (16×16) PEs Intel Touchstone Delta system.

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 [18]. 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. Unlike QR factorization with pivoting, the QR factorization without pivoting and the post- and pre-multiplication by an orthogonal matrix do use blocked operations.

Figure 2 plots the timing results obtained by the PUMMA package using the `BLACS` for the general matrix multiplication, and ScaLAPACK 1.0 (beta version) subroutines for the matrix inversion, QR decomposition with and without column pivoting. Corresponding tabular data can be found in the Appendix.

To measure the efficiency of Algorithm 1, we generated random matrices of different sizes, all of whose entries are normally distributed with mean 0 and variance 1. All computations were performed in real double precision arithmetic. Table 1 lists the measured results of the backward error, the number of Newton iterations, the total CPU time and the megaflops rate. In particular, the second column of the table contains the backward errors and the number of the Newton iterations in parentheses. We note that the convergence rate is problem-data dependent. From Table 1, we see that for a 4000-by-4000 matrix, the algorithm reached 7.19/23.12=31% efficiency with respect to PUMMA matrix multiplication, and 7.19/8.70=82% efficiency with respect to the underlying ScaLAPACK 1.0 (beta) matrix inversion subroutine. As our performance model shows, and tables 9, 10, 11, 12, and 14 confirm, efficiency will continue to improve as the matrix size n increases. Our

Table 1: Backward accuracy, timing in seconds and megaflops of Algorithm 1 on a 256 node Intel Touchstone Delta system.

n	$\Vert E_{21}\Vert_1/\Vert A\Vert_1$ (iter)	Timing (seconds)	Mflops (total)	Mflops (per node)	GEMM-Mflops (per node)	INV-Mflops (per node)
1000	$7.0e-13(18)$	134.22	293.05	1.14	9.04	1.41
2000	$1.6e-12(21)$	448.69	808.28	3.16	15.51	3.88
3000	$3.1e-12(18)$	792.18	1340.60	5.23	19.95	6.43
4000	$5.9e-12(19)$	1436.14	1841.98	7.19	23.12	8.70

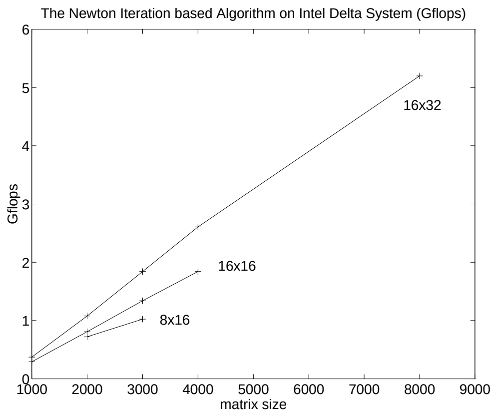

Figure 3: Performance of Algorithm 1 on the Intel Delta system as a function of matrix size for different numbers of processors.

performance model is explained in section 4. Figure 3 shows the performance of Algorithm 1 on the Intel Delta system as a function of matrix size for different numbers of processors.

Table 2 gives details of the total CPU timing of the Newton iteration based algorithm, summarized in Table 1). It is clear that the Newton iteration (sign function) is most expensive, and takes about 90% of the total running time.

To compare with the standard sequential algorithm, we also ran the `LAPACK` driver routine DGEES for computing the Schur decomposition (with reordering of eigenvalues) on one i860 processor. It took 592 seconds for a matrix of order 600, or 9.1 megaflops/second. Assuming that the time scales like n^3, we can predict that for a matrix of order 4000, if the matrix were able to fit on a single node, then DGEES would take 175,000 seconds (48 hours) to compute the desired spectral decomposition. In contrast, Algorithm 1 would only take 1,436 seconds (24 minutes). This is about 120 times faster! However, we should note that DGEES actually computes a complete Schur decomposition with the necessary reordering of the spectrum. Algorithm 1 only decomposes the spectrum along the pure imaginary axis. In some applications, this may be what the users want. If the decomposition along a finer region or a complete Schur decomposition is desired, then the cost of the Newton iteration based algorithms will be increased, though it is likely that the first step just described will

Table 2: Performance Profile on a 256 processor Intel Touchstone Delta system (time in seconds)

n	Sign-Func(%)	QRP(%)	Q^TAQ(%)	total
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

take most of the time [13].

3.2 Implementation and Performance on the CM-5

The Thinking Machines CM-5 was introduced in 1991. The tests in this section were run on a 32 processor CM-5 at the University of California at Berkeley. Each CM-5 node contains a 33 MHz Sparc with an FPU and 64 KB cache, four 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 [53] for more details.

Algorithm 1 was implemented in CM Fortran (CMF) version 2.1 – an implementation of Fortran 77 supplemented with array-processing extensions from the ANSI and ISO (draft) standard Fortran 90 [53]. 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 CM. On receiving an instruction, each node executes it on its own data. When necessary, CM processors can access each other's memory by any of three communication mechanisms, but these are transparent to the CMF programmer [52].

We also used CMSSL version 3.2, [54], TMC's library of numerical linear algebra routines. CMSSL provides data parallel implementations of many standard linear algebra routines, and is designed to be used with CMF and to exploit the vector units.

CMSSL's QR factorization (available 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. Scaling is available to avoid situations when a column norm is close to $\sqrt{\text{underflow}}$ or $\sqrt{\text{overflow}}$, but this is an expensive "insurance policy". Scaling is not used in our current CM-5 code, but should perhaps be made available in our toolbox for the informed user. The QR with pivoting (QRP) factorization routine, which we shall use to reveal rank, 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.

Gaussian elimination with or without partial pivoting is available to compute LU factorizations and perform back substitution to solve a system of equations. Matrix inversion is

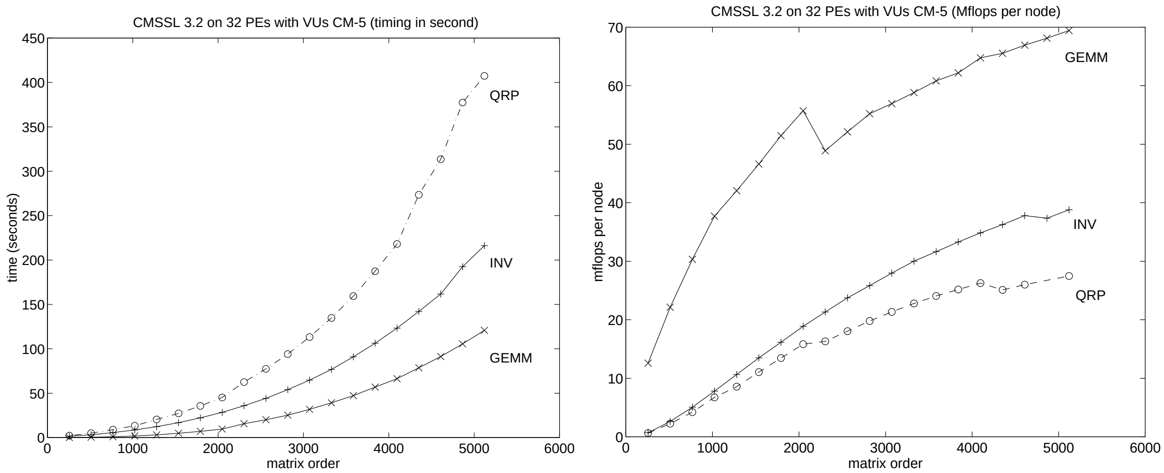

Figure 4: Performance of some CMSSL 3.2 subroutines on 32 PEs with VUs CM-5

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, and which require a determinant computation – and there is a routine for estimating $\|A^{-1}\|_\infty$ from the LU factors to detect ill-conditioning of intermediate matrices in the Newton iteration. Both the factorization and inversion routines balance load by permuting the matrix, and blocking (as specified by the user) is used to improve performance.

The LU, QR and Matrix multiplication routines all have "out-of-core" counterparts to support matrices/systems that are too large to fit in main memory. Our current CM5 implementation of the SDC algorithms does not use any of the out-of-core routines, but in principle our algorithms will permit out-of-core solutions to be used.

Figure 4 summarizes the performance of the CMSSL routines underlying this implementation Algorithm 1.

We tested the Newton-Schulz iteration based algorithm for computing the spectral decomposition along the pure imaginary axis, since matrix multiplication can be twice as fast as matrix inversion; see Figure 4. The entries of random test matrices were uniformly distributed on $[-1, 1]$. We use the inequality $\|A_{i+1} - A_i\|_1 \le \sqrt{n}$ as switching criterion from the Newton iteration (2.5) to the Newton-Schulz iteration (2.6), i.e., we relaxed 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 measured results of the backward accuracy, total CPU time and megaflops rate. The second column of the table is the backward error, the number of Newton iterations and the number of the Newton-Schulz iterations, respectively. From the table, we see that by comparing to CMSSL 3.2 matrix multiplication performance, we obtain 32% to 45% efficiency with the matrices sizes from 512 to 2048, even faster than the CMSSL 3.2 matrix inverse subroutine.

We profiled the total CPU time on each phase of the algorithm, and found that about 83% of total time is spent on the Newton iteration, 9% on the QR decomposition with pivot-

n	$\|E_{21}\|_1/\|A\|_1$ (iter1, iter2)	Actual Time (seconds)	Predicted Time (seconds)	Mflops (total)	Mflops (per node)	GEMM-Mflops (per node)	Inverse-Mflops (per node)
256	$4e-14$(16,2)	25.4	18.9	0.72	0.96	12.57	0.69
512	$2e-13$(16,2)	58.6	50.8	106.88	3.34	22.14	2.62
768	$2e-13$(15,2)	99.23	97.1	203.84	6.37	30.32	5.05
1024	$2e-13$(14,2)	143.92	159.5	318.40	9.95	37.71	7.81
1280	$3e-13$(15,2)	231.12	239.6	405.44	12.67	42.06	10.64
1536	$2e-12$(14,2)	296.99	338.8	520.64	16.27	46.61	13.49
1792	$7e-13$(16,1)	423.42	458.9	579.84	18.12	51.47	16.16
2048	$7e-13$(14,2)	506.11	601.3	732.16	22.88	55.72	18.87

Table 3: Backward accuracy, timing in seconds and megaflops of the SDC algorithm with Newton-Schulz iteration on a 32 PEs with VUs CM-5.

ing, and 7.5% on the matrix multiplication for the Newton-Schulz iteration and orthogonal transformations.

4 Performance 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
b	Block size (in the 2D block cyclic matrix data layout) [20]
τ_{lat}	Time required to send a zero length message from one processor to another.
τ_{band}	Time required to send one double word from one processor to another.
$\tau_{\texttt{DGEMM}}$	Time required per BLAS3 floating point operation

Models for each of the building blocks are given in Table 4. Each model was created by counting the actual operations in the critical path. The load imbalance cost represents the discrepancy between the amount of work which the busiest processor must perform and the amount of work which the other processors must perform. Each of the models for the building blocks were validated against the performance data shown in the appendix. The load imbalance increases as the block size increases.

Because it is based on operation counts, we can not only predict performance, but also estimate the importance of various suggested modifications either to the algorithm, the implementation or the hardware. In general, predicting performance is risky because there are so many factors which control actual performance, including the compiler and various library routines. However, since the majority of the time spent in Algorithm 1 is spent in either the BLACS or the level 3 PB-BLAS[15] (which are in turn implemented as calls to the BLACS[25] and the BLAS[38, 23, 22]), as long as the performance of the BLACS and the BLAS

Task	Computation Cost	Communication Cost		Load Imbalance Cost	
		latency	bandwidth^{-1}	computation	bandwidth^{-1}
LU	$\frac{2}{3}\frac{n^3}{p}\tau_{\texttt{DGEMM}}$	$(6+\lg p)n\tau_{lat}$	$(3+\frac{\lg p}{4})\frac{n^2}{\sqrt{p}}\tau_{band}$	$\frac{bn^2}{\sqrt{p}}\tau_{\texttt{DGEMM}}$	$(1+\frac{\lg p}{4})bn\tau_{band}$
TRI	$\frac{4}{3}\frac{n^3}{p}\tau_{\texttt{DGEMM}}$	$2n\tau_{lat}$	$(2+\frac{3}{2}\lg p)\frac{n^2}{\sqrt{p}}\tau_{band}$	$\frac{2bn^2}{\sqrt{p}}\tau_{\texttt{DGEMM}}$	$\frac{3bn\lg p}{2}\tau_{band}$
Matrix multiply	$2\frac{n^3}{p}\tau_{\texttt{DGEMM}}$	$(1+\frac{\lg p}{2})\sqrt{p}\tau_{lat}$	$(1+\frac{\lg p}{2})\frac{n^2}{\sqrt{p}}\tau_{band}$		
QR	$\frac{4}{3}\frac{n^3}{p}\tau_{\texttt{DGEMM}}$	$3n\lg p\tau_{lat}$	$\frac{3\lg p}{4}\frac{n^2}{\sqrt{p}}\tau_{band}$		
Householder application	$2\frac{n^3}{p}\tau_{\texttt{DGEMM}}$		$2\frac{n^2}{\sqrt{p}}\lg p\tau_{band}$		

Table 4: Models for each of the building blocks

		20 matrix inversions	QR	2 Householder applications	Total
Computation cost	$\times\frac{n^3}{p}\tau_{\texttt{DGEMM}}$	40	$\frac{4}{3}$	4	45
Latency cost	$\times n\tau_{lat}$	$160+20\lg p$	$3\lg p$		$160+23\lg p$
Bandwidth cost	$\times\frac{n^2}{\sqrt{p}}\tau_{band}$	$90+35\lg p$	$\frac{3}{4}\lg p$	4	$90+40\lg p$
Imbalanced computation cost	$\times\frac{bn^2}{\sqrt{p}}\tau_{\texttt{DGEMM}}$	60			60
Imbalanced bandwidth cost	$\times bn\tau_{band}$	$20+35\lg p$			$20+35\lg p$

Table 5: Model of Algorithm 1

are well understood and the input matrix is not too small, we can predict the performance of Algorithm 1 on any distributed memory parallel computer. In Table 5, the predicted running time of each of the steps of Algorithm 1 is displayed. Summing the times in Table 5 yields:

$$\begin{aligned}\text{total time} &= \left(45\frac{n^3}{p}+60\frac{bn^2}{\sqrt{p}}\right)\tau_{\texttt{DGEMM}}+(160+23\lg p)n\tau_{lat}+\\ &\quad \left((90+40\lg p)\frac{n^2}{\sqrt{p}}+(20+35\lg p)bn\right)\tau_{band}. \end{aligned} \tag{4.9}$$

Using the measured machine parameters given in Table 8 with equation (4.9) yields the predicted times in Table 7 and Table 3. To get Table 4 and Table 5 and hence equation (4.9), we have made a number of simplifying assumptions based on our empirical results. We assume that 20 Newton iterations are required. We assume that the time required to send a single message of d double words is $\tau_{lat}+d\tau_{band}$, regardless of how many messages are being sent in the system. Although there are many patterns of communication in the ScaLAPACK implementation, the majority of the communication time is spent in collective communications, i.e. broadcasts and reductions over rows or columns. We therefore choose τ_{lat} and τ_{band} based on programs that measure the performance of collective communications. We assume a perfectly square $\sqrt{p}$-by-$\sqrt{p}$ processor grid. These assumptions allow us to keep the model simple and understandable, but limit its accuracy somewhat.

Table 6: Performance of the Newton iteration based algorithm (Algorithm 1) for the spectral decomposition along the pure imaginary axis, all backward errors $\|E_{21}\|_1/\|A\|_1 \leq 10^{-11}$.

Delta	8×16 PEs			16×16 PEs			16×32 PEs		
n	iter	time (sec)	Mflops (total)	iter	time (sec)	Mflops (total)	iter	time (sec)	Mflops (total)
1000		–	–	18	134.21	293.05	19	110.83	372.94
2000	21	502.57	678.43	21	448.69	808.28	21	336.34	1978.27
3000	18	1037.03	1024.07	18	792.18	1340.60	18	576.68	1841.55
4000		–	–	19	1436.13	1841.98	19	1014.63	2607.18
8000		–	–		–	–	20	4268.35	5197.94

Table 7: Predicted performance of the Newton iteration based algorithm (Algorithm 1) for the spectral decomposition along the pure imaginary axis.

Delta	8×16 PEs		16×16 PEs		16×32 PEs	
n	actual time (sec)	predicted time (sec)	actual time (sec)	predicted time (sec)	actual time (sec)	predicted time (sec)
1000	–	–	134.21	120.1	110.83	112.4
2000	502.57	444.3	448.69	362.3	336.34	310.8
3000	1037.03	994.7	792.18	756.8	576.68	610.4
4000	–	–	1436.13	1334.	1014.63	1026.
8000	–	–	–	–	4268.35	4152.

As Tables 6 and 7 show, our model underestimates the actual time on the Delta by no more than 20% for the machine and problem sizes that we timed. Table 3 shows that our model matches the performance on the CM5 to within 25% for all problem sizes except the smallest, i.e. $n = 256$.

The main sources of error in our model are:

1. uncounted operations, such as small `BLAS1` and `BLAS2` calls, data copying and norm computations,
2. non-square processor configurations,
3. differing numbers of Newton iterations required
4. communications costs which do not fit our linear model,
5. matrix multiply costs which do not fit our constant cost/flop model, and
6. the higher cost of QR decomposition with pivoting.

We believe that uncounted operations account for the main error in our model for small n. The actual number of Newton iterations varies between 18 and 22, whereas we assume exactly 20 Newton iterations are needed. Non-square processor configurations are slightly less efficient than square ones. Actual communication costs do not fit a linear model and depend upon the details such as how many processors are sending data simultaneously and to which processors they are sending. Actual matrix multiply costs depend upon the matrix

Model Parameter	Description	Performance limited by	measured values μs	
			CM5	Delta
$\tau_{\mathtt{DGEMM}}$	BLAS3	peak flop rate	1/90.	1/34.
τ_{lat}	message latency	comm. software	150	157
τ_{band}	bandwidth^{-1}	comm. hardware	1.62	1.67

Table 8: Machine parameters

sizes involved, the leading dimensions and the actual starting locations of the matrices. The cost of any individual call to the BLACS or to the BLAS may differ from the model by 20% or more. However, these differences tend to average out over the entire execution.

Data layout, i.e. the number of processor rows and processor columns and the block size, is critical to the performance of this algorithm. We assume an efficient data layout. Specifically that means a roughly square processor configuration and a fairly large block size (say 16 to 32). The cost of redistributing the data on input to this routine would be tiny, $O((n^2/p)\tau_{band})$, compared to the total cost of the algorithm.

The optimal data layout for LU decomposition is different from the optimal data layout for computing $U^{-1}L^{-1}$. The former prefers slightly more processor rows than columns while the latter prefers slightly more processor columns than rows. In addition, LU decomposition works best with a small block size, 6 on the Delta for example, whereas computing $U^{-1}L^{-1}$ is best done with a large block size, 30 on the Delta for example. The difference is significant enough that we believe a slight overall performance gain, maybe 5% to 10%, could be achieved by redistributing the data between these two phases, even though this redistribution would have to be done twice for each Newton step.

Table 3 shows that except for $n < 512$ our model estimates the performance Algorithm 1 based on CMSSL reasonably well. Note that this table is for a Newton-Shultz iteration scheme which is slightly more efficient on the CM5 than the Newton based iteration. This introduces another small error. The fact that our model matches the performance of the CMSSL based routine, whose internals we have not examined, indicates to us that the implementation of matrix inversion on the CM5 probably requires roughly the same operation counts as the ScaLAPACK implementation.

The performance figures in Table 8 are all measured by an independent program, except for the CM5 BLAS3 performance. The communication performance figures for the Delta in Table 8 are from a report by Littlefield[1] [43]. The communication performance figures for the CM5 are as measured by Whaley[2][58]. The computation performance for the Delta is from the Linpack benchmark[21] for a 1 processor Delta. There is no entry for a 1 processor CM5 in the Linpack benchmark, so $\tau_{\mathtt{DGEMM}}$ in Table 8 above is chosen from our own experience.

5 XI : A Graphical User Interface to SDC

To take advantage of the graphical nature of the spectral decomposition process, a graphical user interface (GUI) has been implemented for SDC. Written in C and based on X11R5's

[1] The BLACS use protocol 2, and the communication pattern most closely resembles the "shift" timings.
[2] τ_{lat} is from Table 8 in[58] and τ_{band} is from Table 5.

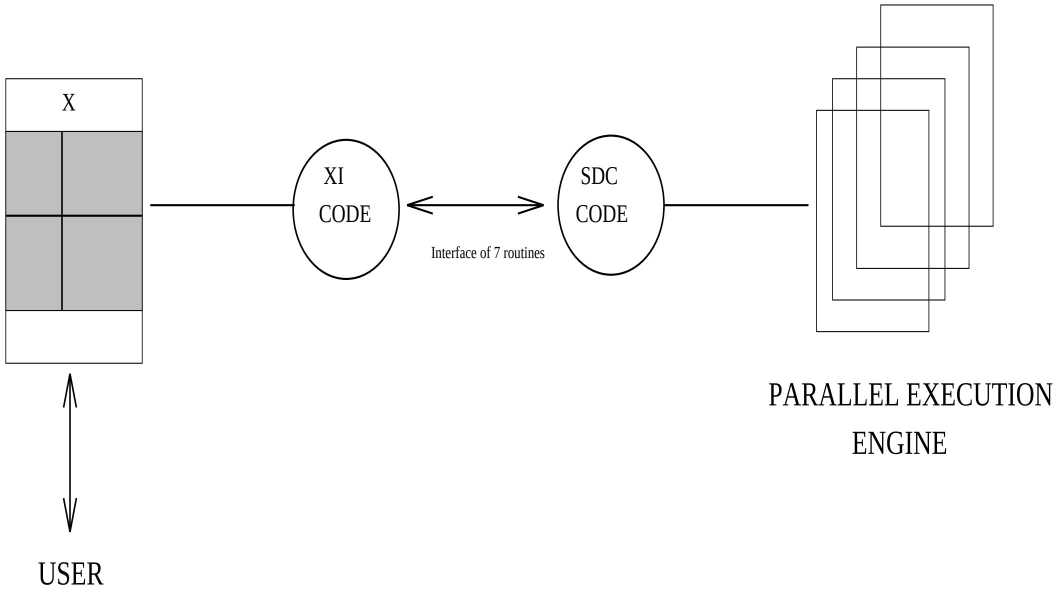

Figure 5: The X11 Interface (**XI**) and SDC

standard Xlib library, the Xt toolkit and MIT's Athena widget set, it has been nicknamed **XI** for "X11 Interface". When **XI** is paired with code implementing SDC we call the union **XSDC**.

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/CMSSL implementation and the Fortran 77 version of our algorithm (both of which use real arithmetic only). Figure 1 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 SDC programmer of these responsibilities and encapsulates the decomposition process. The SDC programmer obtains any useful information via the interface subroutines.

Figure 6 pictures a sample session of *xsdc* on the CM-5 with a 500×500 matrix. The large, 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 matrix under analysis.

Figure 6: A sample *xsdc* session

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 (which can be raised/lowered at the user's request) show the details of the process and will be described later.

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

- `File` lets one save the matrix one is working on, 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 more or less detail 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. This would signal the SDC code to decompose the matrix A to

$$\begin{array}{cc} & \begin{array}{cc} k & n-k \end{array} \\ \begin{array}{c} k \\ n-k \end{array} & \left(\begin{array}{cc} A_{11} & A_{12} \\ 0 & A_{22} \end{array} \right), \end{array}$$

where the k eigenvalues of A_{11} are the eigenvalues of A in the right halfplane, and the eigenvalues of A_{22} are the eigenvalues of A in the left halfplane. The button `Left Halfplane` works similarly, except that the left halfplane would then be selected for further processing and the roles of A_{11} and A_{22} would be reversed. In the same manner, `Inside Circle` and `Outside Circle` divide the complex plane at the boundary of a circle, while `East-West Crosslines` and `North-South Crosslines` split the spectrum with lines at 45 degrees to the real axis (described below).

The `Split Information` window in the lower right corner of Figure 2 keeps track of the matrix splitting process. It reports the two splits performed to arrive at this current (shaded) spectral region. The first, an `East-West Crossline` split at the point 1.5 on the real axis, divided the entire complex plane into four sectors by drawing two lines at ± 45 degrees through the point 1.5 on the real axis. SDC decomposed the starting matrix into:

$$\begin{array}{cc} & \begin{array}{cc} 260 & 240 \end{array} \\ \begin{array}{c} 260 \\ 240 \end{array} & \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \end{array}$$

where the East and West sectors correspond to the A_{11} block while the North and South sectors correspond to the A_{22} block.

Continuing in the East-West sectors as indicated by the previous split, that region is divided into two sub–regions separated by the boundary of the circle of radius 4 centered at the origin. The circle is drawn, making sure that its boundary only intersects the East and West sectors, and the matrix is reduced to:

$$\begin{array}{cc} & \begin{array}{ccc} 106 & 154 & 240 \end{array} \\ \begin{array}{c} 106 \\ 154 \\ 240 \end{array} & \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ 0 & A_{22} & A_{23} \\ 0 & 0 & A_{33} \end{pmatrix} \end{array}$$

The shading indicates that the "bowtie" region (corresponding to the interior of the circle, and the A_{11} block) is currently selected for further analysis.

In the upper right corner of Figure 2 the `Matrix Information` window displays the status of the matrix decomposition process. Each of the three entries corresponds to a spectral region and a square diagonal block of the 3×3 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 along this path of the decomposition process. The highlighted entry corresponds to the shaded region and reports that the A_{11} block contains 106 eigenvalues, has been solved, and is in error by up to 1.44×10^{-13}. The eigenvalues – listed in the window overlapping the Matrix Information window – 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. A click at the point 10 on the imaginary axis for example, would unhighlight the current region and shade the North and South sectors. Since this region corresponds to the A_{33} block, the third entry in the `Matrix-Information` window would be highlighted. The `Split-Information` window would also be updated to detail the single split performed in arriving at this region of the spectrum.

Once a block is small enough, the user may choose to solve it (via the `Function` button at the top of the spectrum window). In this case the eigenvalues, and Schur vectors for that block would be computed using QR (as per the user's request) and the eigenvalues plotted on the spectrum.

The current **XI** code supports real SDC only. It will be extended to handle the complex case as implementations of complex SDC become available.

6 Conclusions and Future work

We have written codes that solve one of the hardest problems of numerical linear algebra: spectral decomposition of nonsymmetric matrices. Our implementation 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 developed in this paper provides a paradigm for us 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 $A - \lambda B$. See [4, 7] for more details.

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 [13] 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.

Our future work will include the implementation and performance evaluation of the inverse free iteration based algorithm, comparison with parallel QR, the extension of the algorithms to the generalized spectral decomposition problem, and the integration of the 3-step approach (see section 2.3) to an object oriented user interface.

Acknowledgements

Bai and Demmel were supported in part by the ARPA grant DM28E04120 via a subcontract from Argonne National Laboratory. 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. Ken Stanley was supported by an NSF graduate student fellowship. Dongarra was also supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-84OR21400.

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, Release 1.0.* SIAM, Philadelphia, 1992. 235 pages.

[2] E. Anderson, Z. Bai, and J. Dongarra. Generalized QR factorization and its applictions. *Lin. Alg. Appl.*, 162–164:243–271, 1992.

[3] L. Auslander and A. Tsao. On parallelizable eigensolvers. *Advances in Applied Mathematics*, 13:253–261, 1992.

[4] Z. Bai and J. Demmel. Design of a parallel nonsymmetric eigenroutine toolbox, Part II. in preparation.

[5] 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).

[6] Z. Bai and J. Demmel. Design of a parallel nonsymmetric eigenroutine toolbox, Part I. In *Proceedings of the Sixth SIAM Conference on Parallel Proceesing for Scientific Computing.* SIAM, 1993. Long version available as UC Berkeley Computer Science report all.ps.Z via anonymous ftp from tr-ftp.cs.berkeley.edu, directory pub/tech-reports/csd/csd-92-718.

[7] Z. Bai, J. Demmel, and M. Gu. Inverse free parallel spectral divide and conquer algorithms for nonsymmetric eigenproblems. Computer Science Division Report UCB//CSD-94-793, UC Berkeley, February 1994. available by anonymous ftp to tr-ftp.cs.berkeley.edu in directory pub/tech-reports/csd/csd-94-79.

[8] 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.

[9] C. Bischof and X. Sun. A divide and conquer method for tridiagonalizing symmetric matrices with repeated eigenvalues. MCS Report P286-0192, Argonne National Lab, 1992.

[10] A. Ya. Bulgakov and S. K. Godunov. Circular dichotomy of the spectrum of a matrix. *Siberian Math. J.*, 29(5):734–744, 1988.

[11] R. Byers. Numerical stability and instability in matrix sign function based algorithms. In C. Byrnes and A. Lindquist, editors, *Computational and Combinatorial Methods in Systems Theory*, pages 185–200. North-Holland, 1986.

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

[13] S. Chakrabarti, J. Demmel, and K. Yelick. On the benefit of mixed parallelism. Computer science division, University of California, 1994. submitted to IPPS.

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

[15] J. Choi, J. Dongarra, and D. Walker. *PB-BLAS: A set of Parallel Block Basic Linear Algebra Subprograms.* University of Tennessee, Knoxville, TN, 1993. available in postscript from netlib/scalapack.

[16] 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).

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

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

[19] 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. NATO-ASI Series E: Applied Sciences, Vol. 232; Available as all.ps.Z via anonymous ftp from tr-ftp.cs.berkeley.edu, in directory pub/tech-reports/csd/csd-92-702.

[20] 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.

[21] J. Dongarra. Performance of various computers using standard linear equations software. Computer science dept. technical report, University of Tennessee, Knoxville, TN, January 1994. up-to-date version available in netlib/benchmark.

[22] 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.

[23] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

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

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

[26] J. Dongarra and D. Walker. The design of linear algebra libraries for high performance computers. Computer Science Dept. Technical Report CS-93-188, University of Tennessee, Knoxville, June 1993. (LAPACK Working Note #58).

[27] 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.

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

[29] 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.

[30] S. K. Godunov. Problem of the dichotomy of the spectrum of a matrix. *Siberian Math. J.*, 27(5):649–660, 1986.

[31] M. Gu and S. Eisenstat. An efficient algorithm for computing a rank-revealing QR decomposition. Computer Science Dept. Report YALEU/DCS/RR-967, Yale University, June 1993.

[32] G. Henry and R. van de Geijn. Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: myths and reality. Computer Science Dept. Technical Report CS-94-244, University of Tennessee, Knoxville, August 1994. (LAPACK Working Note #79).

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

[34] P. Hong and C. T. Pan. The rank revealing QR and SVD. *Math. Comp.*, 58:575–232, 1992.

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

[36] 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 Proceesing for Scientific Computing.* SIAM, 1993.

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

[38] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

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

[40] 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.

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

[42] 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.

[43] 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. Intel Corp.

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

[45] C. Paige. Some aspects of generalized QR factorization. In M. Cox and S. Hammarling, editors, *Reliable Numerical Computations*. Clarendon Press, Oxford, 1990.

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

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

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

[49] 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.

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

[51] G. W. Stewart. Updating a rank-revealing ULV decomposition. *SIAM J. Mat. Anal. Appl.*, 14(2):494–499, April 1993.

[52] Thinking Machines Corporation. *CM Fortran Reference Manual*, December 1992.

[53] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, 1992.

[54] Thinking Machines Corporation. *CMSSL for CM Fortran: CM-5 Edition, version 3.1*, 1993.

[55] R. 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.

[56] 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.

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

[58] R. Clint Whaley. Basic linear algebra communication subroutines: Analysis and implementation across multiple parallel architectures. Technical report, University of Tennessee, Knoxville, June 1994. (LAPACK Working Note #73).

Appendix: Performance data

In this appendix, we record performance data of PUMMA (version 1.0) for matrix multiplication and ScaLAPACK 1.0 (beta version) matrix inversion (LU factorization plus triangular matrix inversion), QR decomposition with and without column pivoting on Intel Touchstone Delta system and performance data of CMSSL 3.2 subroutines for matrix multiplication, matrix inversion, QR decomposition with and without column pivoting on 32 PEs VUs CM-5. Parts of data are used to draw Figures 2 and 4.

Table 9: Performance of PUMMA (version 1.0) matrix multiplication, use `BLACS`, blocksize=10

Delta	16×16 PEs		16×32 PEs	
n	time (seconds)	Mflops (total)	time (seconds)	Mflops (total)
1000	0.86	2314.70	0.62	3237.47
2000	4.03	3971.61	2.57	6236.89
3000	10.57	5108.06	6.49	8320.18
4000	21.63	5918.57	13.61	9406.20
5000	40.27	6207.39	24.54	10187.36
6000	65.80	6565.66	39.52	10930.53
7000	–	–	60.43	11352.03
8000	–	–	87.96	11641.05
9000	–	–	126.01	11570.10

Table 10: Performance of matrix inversion (LU + Triangular inversion), blocksize=30.

Delta	16×16 PEs			16×32 PEs		
n	LU time (seconds)	TRI time (seconds)	Mflops (total)	LU time (seconds)	TRI time (seconds)	Mflops (total)
1000	2.709	2.821	361.67	2.049	2.419	447.63
2000	6.743	9.337	995.01	4.990	7.471	1284.01
3000	12.472	20.460	1639.76	8.971	15.668	2191.66
4000	20.287	37.164	2227.98	14.167	27.431	3077.04
5000	30.676	60.773	2733.78	20.895	43.253	3897.24
6000	44.031	92.136	3172.57	29.258	64.015	4631.52
7000	60.771	132.391	3551.41	39.558	89.561	5312.95
8000	81.240	176.255	3976.78	51.908	117.107	6058.62
9000	106.045	243.910	4166.25	66.579	157.781	6498.48
10000	135.381	316.825	4422.77	83.737	202.448	6988.48

Table 11: Performance of QR decomposition method for solving the least squares problem (QR decomposition + Triangular solver), blocksize=6.

Delta	16×16 PEs			16×32 PEs		
n	QR time (seconds)	Solve time (seconds)	Mflops (total)	QR time (seconds)	Solve time (seconds)	Mflops (total)
1000	4.514	1.0033	242.77	3.440	0.3975	348.26
2000	11.495	2.1829	781.61	8.381	0.8433	1157.63
3000	21.791	3.6266	1418.44	15.414	1.3375	2150.73
4000	36.150	5.3768	2057.21	24.752	1.8519	3209.40
5000	55.513	7.2712	2656.98	36.911	2.3994	4241.71
6000	80.682	9.5162	3195.36	52.452	3.0455	5191.39
7000	112.359	12.0318	3678.94	71.531	3.6736	6083.12
8000	151.524	14.7269	4108.54	94.682	4.3310	6896.66
9000	198.830	17.8164	4488.82	122.305	5.0823	7632.18
10000	255.710	20.9755	4821.16	154.792	5.8376	8302.54

Table 12: Performance of QR decomposition with column pivoting.

Delta	16×16 PEs		16×32 PEs	
n	time (seconds)	Mflops (total)	time (seconds)	Mflops (total)
1000	6.6254	201.25	4.6249	288.29
2000	18.1142	588.86	11.8456	900.48
3000	35.8709	1003.60	22.3526	1610.55
4000	62.3064	1369.58	37.2151	2292.98
5000	98.1878	1697.43	57.3836	2904.43
6000	147.1279	1957.48	83.2704	3458.61
7000	209.4396	2183.61	116.7009	3918.85
8000	286.0408	2386.61	157.7757	4326.82
9000	–	–	207.2685	4689.57
10000	–	–	265.6364	5019.39

Table 13: Backward accuracy, timing in seconds and megaflops of the SDC algorithm with Newton iteration on a 32 PEs with VUs CM-5.

n	$\|E_{21}\|_1/\|A\|_1$ (iter)	Timing (seconds)	Mflops (total)	Mflops (per node)	GEMM-Mflops (per node)	INV-Mflops (per node)
256	$4e-14(18)$	33.3	21.76	0.68	12.57	0.69
512	$2e-13(18)$	67.7	84.48	2.64	22.14	2.62
768	$3e-13(17)$	115.1	160.00	5.00	30.32	5.05
1024	$3e-13(16)$	165.0	251.52	7.86	37.71	7.81
1280	$4e-13(18)$	259.1	329.28	10.29	42.06	10.64
1536	$2e-12(16)$	331.4	422.72	13.21	46.61	13.49
1792	$9e-13(17)$	456.9	512.32	16.01	51.47	16.16
2048	$1e-12(16)$	552.0	601.60	18.80	55.72	18.87

Table 14: Performance of matrix multiplication and matrix inversion, and QRP, CMSSL version 3.2

CM-5	GEMM		Inversion		QRP	
n	time (seconds)	Mflops (per node)	time (seconds)	Mflops (per node)	time (seconds)	Mflops (per node)
256	0.08	12.57	1.51	0.69	2.23	0.63
512	0.38	22.14	3.21	2.62	5.01	2.23
768	0.93	30.32	5.60	5.05	8.96	4.21
1024	1.78	37.71	8.59	7.81	13.29	6.73
1280	3.12	42.06	12.32	10.64	20.41	8.56
1536	4.86	46.61	16.80	13.49	27.32	11.05
1792	6.99	51.47	22.26	16.16	35.65	13.45
2048	9.64	55.72	28.45	18.87	45.21	15.83
2304	15.64	48.89	35.86	21.32	62.51	16.30
2560	20.12	52.12	44.17	23.74	77.45	18.05
2816	25.28	55.22	54.03	25.83	94.06	19.78
3072	31.82	56.94	64.77	27.98	113.20	21.34
3328	39.16	58.83	76.76	30.01	134.75	22.79
3584	47.30	60.83	90.97	31.63	159.38	24.07
3840	56.91	62.19	106.30	33.29	187.37	25.18
4096	66.32	64.76	123.23	34.85	218.05	26.26
4352	78.60	65.55	142.09	36.26	273.52	25.11
4608	91.35	66.94	161.77	37.80	313.53	26.00
4864	105.56	68.13	192.56	37.35	377.39	
5120	120.80	69.44	216.21	38.80	407.42	27.47
5376	–	–	–	38.80	493.26	26.25
5632	–	–	–	38.80	554.36	26.85
5888	–	–	–	38.80	628.69	27.06
6144	–	–	–	38.80	699.88	27.62