COMPUTING EIGENVALUES AND EIGENVECTORS OF A DENSE REAL SYMMETRIC MATRIX ON THE NCUBE 6400

S. Chinchalkar and T. F. Coleman

June 1991

1 Introduction

This report demonstrates parallel versions of the Eispack [Smith 76] functions TRED2 and TQL2 for finding all eigenvalues and eigenvectors of a dense, real symmetric matrix on the Ncube 6400. The real symmetric eigenvalue problem can be posed as follows:
Given a real symmetric $n \times n$ matrix A, compute a diagonal matrix D and an orthonormal matrix Q such that:

$$A = QDQ^T$$

The diagonal elements of D are the eigenvalues of A and the columns of Q are the eigenvectors of A.

There are several techniques for solving the above problem. An efficient and accurate method is tridiagonalization of the original matrix A (TRED2), followed by application of the QR method on the resulting tridiagonal matrix (TQL2). Since the eigenvalues are the roots of the characteristic polynomial $|A - \lambda I| = 0$, bisection and inverse iteration can be used to compute all eigenvalues and eigenvectors. This method is significantly faster than the QR method provided the eigenvalues are well separated. If the eigenvalues are not well separated, the resulting eigenvectors need to be re-orthogonalized, requiring more computational effort. The numerical accuracy is also not as high as the QR method. Recently, Cuppen's divide and conquer algorithm was shown to be significantly faster than the QR method on sequential machines [Dongarra 87]. However, when the entire eigensystem (all eigenvalues and eigenvectors) is desired, the QR method parallelizes more easily than the divide and conquer method. Firstly, one of the reasons why the divide and conquer method is faster than the QR algorithm on a sequential computer is because of 'deflation', which reduces the size of the subproblems. On multiple processors, the benefits of deflation are not shared equally by all processors, resulting in a loss of efficiency [Ipsen 88]. Secondly, this algorithm requires considerable interprocessor communication, whereas the QR algorithm requires none. This makes the divide and conquer algorithm more difficult to parallelize than the QR algorithm. It is still possible (and quite likely) for the divide and conquer algorithm to out-perform the QR algorithm in parallel because it is inherently faster.

The next few sections describe the algorithms TRED2 and TQL2 and their parallel counterparts in greater detail. The results of numerical experiments on large problems on a 1024 processor Ncube 6400 are also presented.

2 The Ncube 6400

The Ncube 6400 (also known as the Ncube 2) is a distributed memory MIMD machine with up to 8192 node processors and 1 host processor. Each node processor is a 64-bit chip supporting up to 64 Mbytes of memory [Dunigan 91]. The programs written as a part of this work were tested on a 1024 processor machine with 4 Mbytes of memory per processor. High performance BLAS run at an asymptotic speed of about 1.8-2.0 Mflops (depending on the subprogram) per processor. Data is sent from one processor to another by explicit message-passing. Nearest neighbor communication takes approximately $200 + 0.6N$ microseconds for a message of N bytes [Dunigan 91].

The Portable Instrumented Communication Library (PICL) [Geist 91] was used to write portable host and node programs. The use of this software library relieves the programmer of the additional burden of learning a new parallel programming environment. PICL also contains a set of higher level global communication routines for efficient broadcast, global sum and so on. The overhead in using this library as compared to the native Ncube communication utilities is typically small [Geist 90].

3 Algorithms

This section describes the sequential and parallel algorithms used for solving the real symmetric eigenvalue problem.

3.1 TRED2

TRED2 is the Eispack subroutine for tridiagonalizing a real symmetric matrix using Householder reductions. Given a real symmetric $n \times n$ matrix A, TRED2 computes Q_1 and T such that

$$Q_1^T A Q_1 = T$$

where Q_1 is orthonormal and T is tridiagonal.

A is reduced to T by the application of $n - 2$ Householder transformations. Q_1 is initialized to the identity matrix and formed by backward accumulation of the Householder transformations, i.e., the Householder transformations are applied to the orthonormal matrix in reverse order (the first Householder transformation applied to A is the last one applied to Q_1). On a sequential machine, the Householder vectors are stored in the matrix A itself. After the reduction to tridiagonal form is completed, these vectors are used in reverse order to form Q_1 explicitly. However, on a parallel machine, the matrix A is distributed among the processors. Therefore, the Householder vectors would also have to be stored in a distributed fashion. During the formation of Q_1, all processors need all Householder vectors. Therefore, formation of Q_1 would require considerable communication overhead. Moreover, when Q_1 is formed by backward accumulation, it is natural to distribute it by columns. It is shown in the next subsection that in the QR algorithm, it is more natural to distribute Q_1 by rows. If that is the case, a parallel matrix transpose is necessary. Instead, by applying the Householder transformations to the orthonormal matrix in the same order in which they are generated, the communication overhead can be avoided. This method of forming Q_1 is called forward accumulation. The penalty for using forward accumulation is not very high, as shown below.

The computational complexity in terms of the number of floating point operations (flops)[1] of various stages of the algorithms are given below. Note that the QR algorithm is an iterative algorithm and the number of floating point operations given is what is typically observed.

Householder reductions	$= 4n^3/3$
Backward accumulation	$= 4n^3/3$
Forward accumulation	$= 2n^3$
QR algorithm	$= 6n^3$

Therefore, if backward accumulation is used, the total computational cost is $26n^3/3$ flops as compared to $28n^3/3$ flops when forward accumulation is used. This is an increase of only 7.7%; much less than typical communication overhead.

Figure 1 shows the serial algorithm. In the parallel case, a wrap mapping[2] is used to distribute the rows of A across the hypercube. Each processor, in turn, computes the Householder vector corresponding to the rows of A that belong to it, and broadcasts it to all other processors. Each processor then applies the Householder transformation to the remaining portion of A with it, and to its rows of Q_1. Upon termination, all processors have the tridiagonal matrix T, and the rows of Q_1 are distributed among the processors in a wrap fashion. The parallel algorithm is given in Figure 2.

3.2 TQL2

TQL2 is the Eispack subroutine which uses the explicit QR algorithm for diagonalizing a given tridiagonal matrix. Given a tridiagonal matrix T of size n, and an $n \times n$ orthonormal matrix Q_1, TQL2 computes an $n \times n$ orthonormal matrix Q and a diagonal matrix D of size n such that

$$\begin{aligned} Q_2^T T Q_2 &= D \\ Q &= Q_1 Q_2 \end{aligned}$$

where Q_2 is an $n \times n$ orthonormal matrix.

Figure 3 describes 1 sweep of the serial QR algorithm. It is important to note that most of the computational effort ($O(n^3)$) is spent in applying the Givens rotations to the eigenvector matrix; the calculation of eigenvalues is low order work ($O(n^2)$).

In the parallel case (Figure 4), the orthonormal matrix Q_1 is distributed by rows (Q_1 is obtained from TRED2) among the processors. Each processor computes all rotations and applies them to the tridiagonal matrix T. Each processor also applies the same rotations to all rows of Q_1 that belong to it. Upon termination, T is transformed to D, whose diagonal entries are eigenvalues of the original matrix A and Q_1 is transformed into Q. All processors have a copy of D, whereas Q is distributed by rows among the processors in exactly the same manner as the input matrix Q_1. The

[1] $y \leftarrow y + a * x$ is considered as 2 flops

[2] row i belongs to processor i **mod** p, where p is the number of processors

Input : $A_{n\times n}$ real symmetric
Output : Q_1 and T such that $T = Q_1^T A Q_1$ where T is tridiagonal and Q_1 is orthonormal. T is stored in two vectors; t_d contains the diagonal and t_u contains the upperdiagonal.

$Q_1 = I$
for $k = 1 : n-2$
  $u = A(k, k+1 : n)^T$
  $v = u + \|u\|_2 \texttt{sign}(u_1) e_1$
  $p = 2A(k+1 : n, k+1 : n)v/v^T v$
  $w = p - (p^T v)v/v^T v$
  $A(k+1 : n, k+1 : n) = A(k+1 : n, k+1 : n) - vw^T - wv^T$
  $y = -2Q_1 v$
  $Q_1 = Q_1 + yv^T$
end
$t_d = \texttt{diagonal}(A)$
$t_u = \texttt{upperdiag}(A)$

Figure 1: Sequential TRED2 algorithm

matrix Q_2, which can be thought of as the product of all orthonormal rotations, is not constructed explicitly. It is easy to see that this algorithm does not require any interprocessor communication.

4 Implementation Details

The above algorithms hide several details for clarity of notation. They are, however, very important when implementing the algorithm on a parallel machine. Symmetry of A and T is exploited in order to reduce computational effort during updates as well as the memory requirements. The matrices T and D are stored as vectors instead of full matrices. The storage requirement of the eigensolver is $O(n^2/p)$, where n is the problem size and p is the number of processors. In order to distribute data most efficiently among the nodes of the hypercube, minimum spanning trees are used when performing a broadcast or a global sum.

During the tridiagonalization stage, if $A(k, k+1 : n)$ is "small", the corresponding Householder transformation is skipped. In the QR algorithm, deflation can occur if an off-diagonal element of T is sufficiently small. If this is the case, the matrix is split into two smaller matrices and the QR sweep is applied separately to each of the two smaller matrices. The Eispack subroutine TRED2 uses the explicit QR algorithm; in practice, the implicit QR algorithm is slightly more stable. However, the parallelization issues in both the cases are identical.

The efficiency of TRED2 can be increased by 'pipelining' [Geist 86]. Tridiagonal-

Input : $A_{n\times n}$ real symmetric matrix, distributed by rows across p processors of the hypercube in a wrap fashion.
Output : Q_1 and T such that $T = Q_1^T A Q_1$ where T is tridiagonal and Q_1 is orthonormal. All processors have T and the rows of Q_1 are distributed across p processors in a wrap fashion. T is stored in two vectors; t_d contains the diagonal and t_u contains the upperdiagonal.

```
Q_1 = I
for k = 1 : n-2
    if (I have row k)
        u = A(k, k+1 : n)^T
        v = u + ||u||_2 sign(u_1) e_1
        send v to all nodes
    else
        receive v
    endif
    for j = k+1 : n
        if (I have row j)
            p_i(j) = 2A(j, k+1 : n)v/v^T v
        endif
    end
    p = gather(p_i, allnodes)
    w = p - (p^T v)v/v^T v
    for j = k+1 : n
        if (I have row j)
            A(j, k+1 : n) = A(j, k+1 : n) - v(j)w^T - w(j)v^T
        endif
    end
    for j = 1 : n
        if (I have row j)
            y = -2Q_1(j, 1 : n)v/v^T v
            Q_1(j, 1 : n) = Q_1(j, 1 : n) + yv^T
        endif
    end
end

t_{d_i} = diagonal(A)
t_{u_i} = upperdiag(A)
t_d = gather(t_{d_i}, allnodes)
t_u = gather(t_{u_i}, allnodes)
```

Figure 2: Parallel TRED2 Algorithm

Input : Symmetric tridiagonal matrix T and an orthonormal matrix Q_1.
Output : D and Q such that D is diagonal, Q is orthonormal and $Q_2^T T Q_2 = D$ $Q = Q_1 Q_2$ where Q_2 is orthonormal.
A single sweep looks like :

calculate shift μ
$T = T - \mu I$
`for` $i = n-1 : -1 : 1$
  compute $P^i_{2\times 2}$ in order to rotate rows i and $i+1$ of T suitably.
  $T = P^{i^T} T P^i$
  $Q = QP^i$
`end`
$T = T + \mu I$

Figure 3: Sequential TQL2 Algorithm

Input : Each processor has a copy of the symmetric tridiagonal matrix T and Q_1 is an orthonormal matrix distributed by rows.
Output : D and Q such that D is diagonal, Q is orthonormal and $Q_2^T T Q_2 = D$ $Q = Q_1 Q_2$ where Q_2 is orthonormal. Each processor has D and Q is distributed by rows.
A single sweep looks like :

calculate shift μ
$T = T - \mu I$
`for` $i = n-1 : -1 : 1$
  compute $P^i_{2\times 2}$ in order to rotate rows i and $i+1$ of T suitably.
  $T = P^{i^T} T P^i$
  `for` $k = 1 : n$
    `if (I have row` k`)`
      $Q(k, i : i+1) = Q(k, i : i+1) P^i$
    `endif`
  `end`
`end`
$T = T + \mu I$

Figure 4: Parallel TQL2 Algorithm

ization consists of two steps, computation of the Householder vector and application of the Householder transformation to the remaining portion of the matrix (and the eigenvector matrix). They can be represented concisely as

```
for i = 1 to n − 2
      compute_vec(i)
      for k = i + 1 to n
           update(k, i)
      end
end
```

where `compute_vec(i)` computes the Householder vector based on row i of A and the vectors p and w and `update(k, i)` updates row k of A using the Householder vector based on row i. As soon as all updates to row k are completed, the Householder vector based on row k can be computed and broadcast to all other processors before continuing with the rest of the updates. This reduces the amount of time other processors have to wait for the k^{th} Householder vector, thus increasing the overall efficiency.

In order to test the accuracy of the results, $\|QQ^T - I\|_F$ and $\|QDQ^T - A\|_F$ were computed in parallel by the node processors. Since the parallel algorithms are essentially the same as their serial counterparts, the numerical robustness is unaffected.

5 Numerical Experiments

This section presents the results of numerical experiments performed on the Ncube. Efficiency of parallelization and Mflop rate for problems of different sizes on cubes with different number of processors are shown. All tests were performed on random matrices.

As mentioned earlier, the number of iterations needed for the QR algorithm is not known a-priori. Therefore, in order to compute the number of floating point operations performed, the program was made to print r, the total number of Givens rotations applied to the tridiagonal matrix T. The number of floating point operations per rotation is $6n$. Therefore, the total number of flops for the entire solution is given by:

$$\text{total flops} = \frac{10n^3}{3} + 6nr$$

Speedup is defined as:

$$\text{speedup} = \frac{\text{time taken by program on 1 processor}}{\text{time taken by program on } p \text{ processors}}$$

Efficiency of parallelization is defined as:

$$\text{efficiency} = \frac{\text{speedup}}{p} \times 100$$

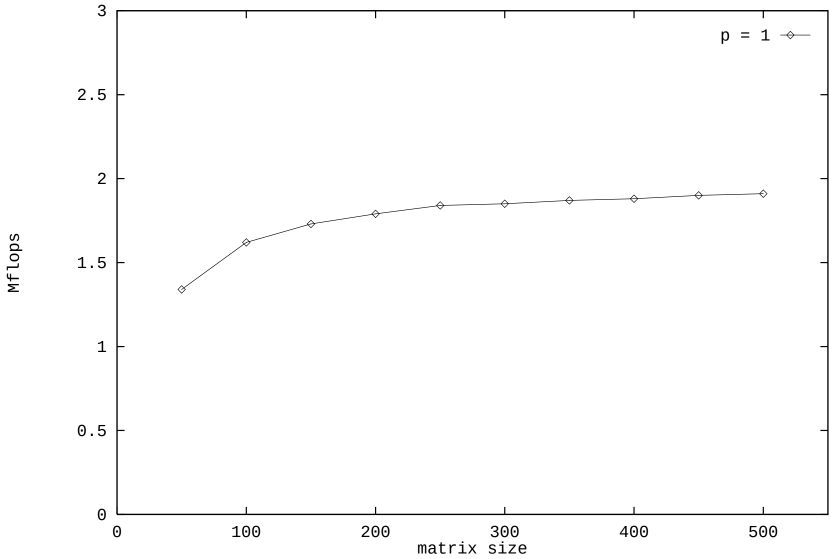

Figure 5: Speed of program on 1 processor

It is important to note that in order to determine the time taken by the program on 1 processor, a separate program was written which did not require any parallel overhead (such as extra conditional statements).

Since it is not possible to solve large problems on 1 processor, in order to compute the efficiency of parallelization by the above formula, the asymptotic speed of 1 processor was estimated by solving eigenvalue problems of increasing size on 1 node of the hypercube and determining the execution speed. Figure 5 shows the variation in speed of 1 processor with problem size. It can be seen that the speed increases initially because the vectors being operated upon by BLAS become longer. But eventually, the speed levels off at around 1.92 Mflops. This speed, along with the formula for number of flops was used to estimate the time taken by the program on 1 processor for large problems.

Figures 6 and 7 show the variation in efficiency of parallelization with respect to the number of processors p and the matrix size n respectively. As expected, the efficiency increases with an increase in problem size because the computational effort per processor increases, resulting in lower percentage communication overhead.

Figure 8 shows the variation in speed as a function of problem size for cubes of size 128, 256 and 512. It can be seen that for the largest problem solved on a 9-dimensional cube ($n = 11000$, $p = 512$), the speed was 539 Mflops.

6 Conclusion

This report demonstrates fast, parallel implementations of the Eispack routines TRED2 and TQL2 for computing all eigenvalues and eigenvectors of a real symmet-

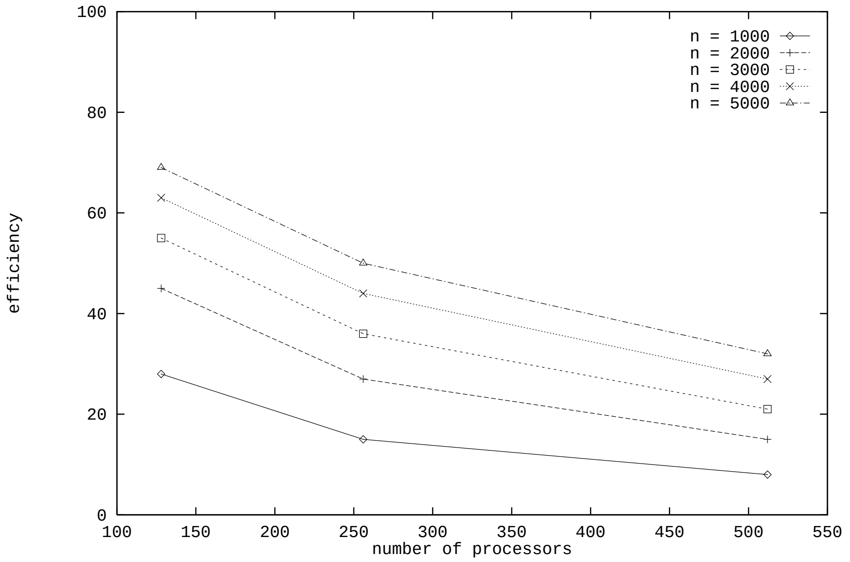

Figure 6: Variation in efficiency as a function of number of processors

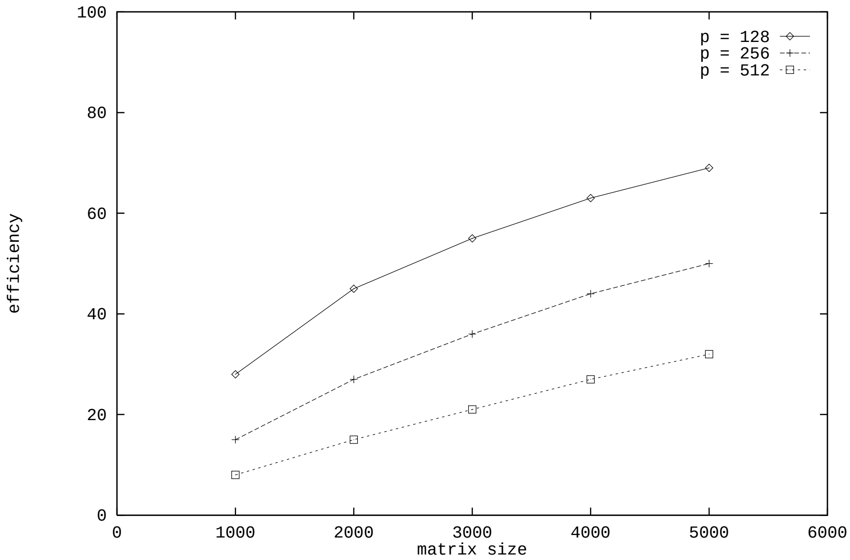

Figure 7: Variation in efficiency as a function of matrix size

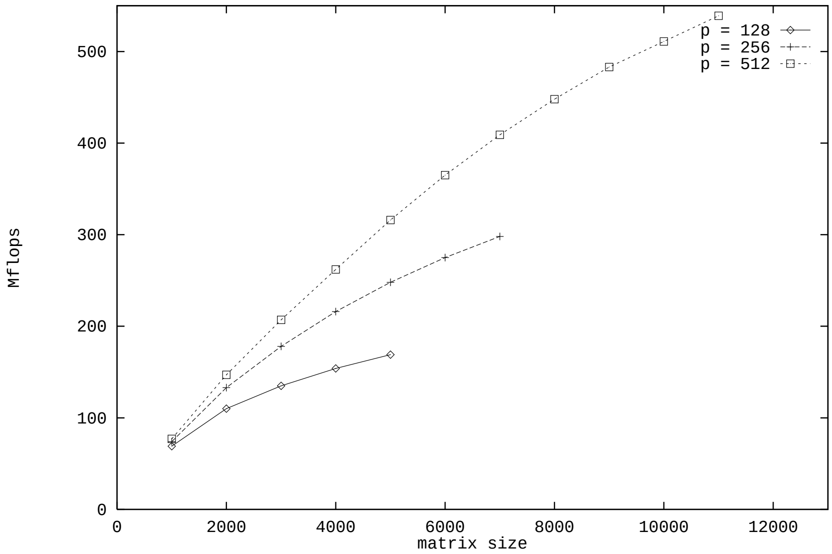

Figure 8: Variation in program speed as a function of problem size

ric matrix on distributed memory multiprocessors. Numerical experiments performed on an Ncube 6400 show that high performance can be obtained, while retaining the numerical robustness of the original sequential code.

7 Acknowledgment

All numerical experiments were performed on an Ncube 6400 operated by the Sandia National Laboratory.

References

[Dongarra 87] Dongarra, J. J. and Sorensen, D. C., "A Fully Parallel Algorithm for the Real Symmetric Eigenvalue Problem", *SIAM Journal for Scientific and Statistical Computing*, V8, N2, March 1987.

[Dunigan 91] Dunigan, T. H., "Performance of the Intel iPSC/860 and Ncube 6400 Hypercubes", ORNL/TM-11790, Oak Ridge National Laboratory, April 1991.

[Geist 86] Geist, G. A. and Heath, M. T., "Matrix Factorization on a Hypercube Multiprocessor", *Hypercube Multiprocessors 86*, SIAM, pp. 161-180, 1986.

[Geist 91] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H., "A Users' Guide to PICL: A Portable Instrumented Communication Library", ORNL/TM-11616, Oak Ridge National Laboratory, February 1991.

[Geist 90] Geist, G. A., Heath, M. T., Peyton, B. W., and Worley, P. H., "PICL: A Portable Instrumented Communication Library", ORNL/TM-11130, Oak Ridge National Laboratory, July 1990.

[Golub 89] Golub, G. H. and Van Loan, C. F., "Matrix Computations", Second Edition, *Johns Hopkins University Press*, 1989.

[Ipsen 88] Ipsen, I. C. F. and Jessup, E. R., "Solving the Symmetric Tridiagonal Eigenvalue Problem on the Hypercube", Research Report YALEU/DCS/RR-548, Department of Computer Science, Yale University, 1988.

[Smith 76] Smith, B. T., Boyle, J. M., Dongarra, J. J., Garbow, B. S., Ikebe, Y., Klema, V. C. and Moler, C. B., "Matrix Eigensystem Routines - EISPACK Guide", *Lecture Notes in Computer Science*, 6, 2^{nd} ed., Springer Verlag, Berlin, 1976.