An Efficient Parallel Version of the Householder-QL Matrix Diagonalisation Algorithm

J.S. Reeve*and M. Heath
Department of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ, UK
email jsr@ecs.soton.ac.uk

October 9, 1998

Abstract

In this paper we report an effective parallelisation of the Householder routine for the reduction of a real symmetric matrix to tri-diagonal form and the QL algorithm for the diagonalisation of the resulting matrix. The Householder algorithm scales like $\alpha N^3/P + \beta N^2 \log_2(P)$ and the QL algorithm like $\gamma N^2 + \delta N^3/P$ as the number of processors P is increased for fixed problem size. The constant parameters α, β, γ and δ are obtained empirically. When the eigenvalues only are required the Householder method scales as above while the QL algorithm remains sequential. The code is implemented in c in conjunction with the Message Passing Interface (MPI) libraries and verified on a sixteen node IBM SP2 and for real matrices that occur in the simulation of properties of crystaline materials

Keywords **Householder; QL; Parallel Algorithms.**

1 Introduction

The parallelisation of methods for finding the eigenvalues and eigenvectors for real symetric matrices has received little attention over the years, for two reasons. Firstly for very large systems quite often only a few eigenvalues and eigenvectors are needed. Secondly, the parallelisation of the best sequential techniques for matrix diagonalisation has proved difficult and ineffective.

In the simulation of the dynamics of large structures, only the lowest few eigenvalues that correspond to the most likely occuring low frequency modes of vibration are needed. Similarly, in solid state physics, states at or near the ground state are those most likely to be occupied, at least at low temperatures, and so make the largest contribution to the material properties. For these reasons, in subjects like structural mechanics, techniques have concentrated in reducing the degrees of freedom in the system, from perhaps millions to a few hundred, enabling classical sequential techniques to be used. In Physics and Chemistry, and where the original matrix is regularly structured and sparse, iterative techniques like the Lanczos[9] or Davidson[2] methods, that recover the lowest few eigenvalues and their corresponding eigenvectors can be used effectively.

There are however many problems for which it is desirable to have all of the eigenvalues and eigenvectors of systems. Physical simulations of quantum mechanical systems that have a few particles present at finite temperature cannot be properly analysed by methods for large numbers of particles like statistical mechanics and are sufficiently far from their ground states as to be influenced by all available states. Similarly in the design of mechanical structures acoustic resonances might occur over a broad bandwidth. In such cases all eigenvalues and eigenvectors of large symmetric matrices are needed, for which no algorithm of complexity less than $O(N^3)$, where N is the dimension of the matrix to be diagonalised, exists, regardless of the sparsity of the matrix.

The most efficient algorithm for the diagonalisation of large symmetric matrices is the Householder tri-diagonalisation proceedure followed by the QL diagonalisation algorithm. The operation count for the combined Householder method is approximately N^3 whether or not eigenvectors are required and the operation count for the QL method is roughly $30N^2$ when the eigenvectors are not required and $3N^3$ when they are. These counts presume that the average number of iterations is about 1.7[10], an assumption that is confirmed for the test matrices that we use. The only viable alternative is the Jacobi method which has about $20N^3$ operations when eigenvectors are not required and about $40N^3$ when the are assuming less that 10 Jacobi cycles for convergence[10]. The Jacobi method might readily parallelise [11] but requires more than 20 processors before becoming competitive with the sequential Householder-QL algorithm. Previous attempts at parallelising the Householder and QL algorithms have resulted in inefficient solutions[4] or have altered the algorithms in some way[12, 3, 1, 8]

2 Description of the Algorithm

Clear descriptions of the Householder and QL algorithms can be found in [10] and [6]. If A is an $N \times N$ symmetric matrix, then a sequence of $N-2$ transformations of the form $P_{N-2}...P_1AP_1...P_{N-2}$ reduces the matrix to symmetric tri-diagonal form. The matrix P_k is formed from the requirement that the matrix $A_{k-1}= P_{k-1}...P_1AP_1...P_{k-1}$, is tri-diagonal up to the kth column. For example for a 6×6 matrix A_1 has the form

$$\begin{pmatrix} d_1 & e_1 & 0 & 0 & 0 & 0 \\ e_1 & d_2 & x & x & x & x \\ 0 & x & x & x & x & x \\ 0 & x & x & x & x & x \\ 0 & x & x & x & x & x \\ 0 & x & x & x & x & x \\ 0 & x & x & x & x & x \end{pmatrix}$$

The P_k matrices can be written as $P_k = I - 2W_kW_k^T$, where the vector $W_k^T = R^{-1}(0,\ldots,0,x_{k+1}+s,x_{k+2},\ldots,x_n)^T$, is formed from the kth row (or column) of the A_{k-1} matrix, where R normalises W_k and $s = sign(x_{k+1})(x_{k+1}^2 + x_{k+2}^2 + \ldots + x_n^2)$. The A_k matrix is then calculated as $A_k = P_kA_{k-1}P_k$ which can readily be shown to reduce to $A_k = A_{k-1} - 2W_kQ_k^T - 2Q_kW_k^T$, where $Q_k = V_k - cW_k$, with $V_k = A_{k-1}W_k$ and $c = W_k^TV_k$.

The complete transformation matrix $Z = P_{N-2}...P_1$ is needed for the calculation of the eigenvectors. The normal method for reconstructing Z is to store the W_k vectors in the rows of the A matrix that have already been tri-diagonalised, then the Z matrix can be built using the same space so that all the eigenvectors of A can be determined using only the memory occupied by the original data space.

The QL algorithm which is used to find all the eigenvalues (and eigenvectors) of a symmetric tri-diagonal matrix naturally follows the Householder transformation. The part of the QL algorithm that generates the eigenvalues has complexity $O(N^2)$, and the part that builds the eigenvectors has complexity $O(N^3)$, so that it is only worthwhile parallelising the QL algorithm if the eigenvectors are required. The code fragment of the QL algorithm that constructs the eigenvectors has the form below for the Ith iteration:

```
FOR(J=0;J<N;J++){
    F=Z[I+1][J]
    Z[I+1][J]=S*Z[I][J]+C*F
    Z[I][J]=C*Z[I][J]-S*F
}
```

When the transformation matrix Z is stored by row-wise striping[5] the update of the Z matrix can be performed independently for each processor, except when the Ith and I+1th rows are on different processors. As the iteration sequence is from N to 1, at most a single exchange of rows is needed for each iteration. This dictates that the transformation matrix from the Householder algorithm also be decomposed by row-wise striping, which in turn means that we have to use the same storage method for the original matrix. In fact the only significant amount of memory needed is that occupied by the original matrix, assuming of course that this is stored as a dense matrix. This important property needs to be retained in the parallel algorithm.

In the mathematical description of the Householder algorithm that follows the iteration number is indicated by subscripting the vectors, for example W_k is the W vector at the kth iteration. In the pseudo code there is only one storage location for each vector and in the case of the W_k, it is denoted as W. The iteration number in the pseudo code is K.

The pseudo-code outline for the Householder routine is:-

```
for(k=0;k<N-2;k++){
    s=formW(W,A,k)
    formV(V,A,W)
    formQ(Q,V,W,s,k)
    formA(A,Q,W,s,k)
}
formZ(A)
```

As the matrix A_k is stored row-wise the formation of the W_k is done by one processor and broadcast to the rest. The quantity s, which is needed for subsequent parts of the calculation is also distributed in this same broadcast. The pseudo-code outline for formW is :-

```
double formW(W,A,k) –Inputs A_{k-1} and outputs W_k
    if(ProcNoForRow[k] == MyProcNo){
        form W and s
        MPI_Bcast(s,W)
    }
    else
        MPI_Bcast(s,W)
    }
    return(s)
```

Calculation of the vector $V_k = A_k W_k$ requires a matrix vector product. Since each processor has a copy of the W_k vector, this requires a local matrix vector product followed by a gathering operation of the local slices of V_k, so that each processor has a complete copy of V_k.

```
formV(V,A,W,k) –Inputs A_{k-1} and W_k and outputs V_k
    Do the local matrix multiply V_l = AW
    MPI_Allgather(V_l,V)
```

As both the vectors V_k and W_k are held by each processor the Q_k vector (calculated in formQ) can be computed locally on each processor. The A_k matrix is updated by the equation $A_k = A_{k-1} - 2W_k Q_k^T - 2Q_k W_k^T$, which involves only outer products of the locally stored W_k and Q_k vectors. The diagonal and off-diagonal elements of the reduced part of the a matrix are stored in a local vector on the processor that holds row k. At the end of the tri-diagonalisation process these need to be gathered onto the other processors so all processors have a copy of the tri-diagonalised matrix as required by the QL procedure. If the eigenvectors are not required then the transformation matrix $Z = P_{N-2}, \ldots, P_1$ is not required and FORMZ can be omitted. When the eigenvectors are required, Z is updated recursively from the formula $Z_k = Z_{k-1} - 2Z_{k-1}W_kW_k^T$. The formation of Z starts from the bottom right hand corner of the original A matrix, which now stores the W_k row-wise. Proceeding from this corner for which Z is the 2×2 unit matrix, we can form the Z_k matrix in rows k up to N in the space formerly used to keep the W_k up to W_{N-2}. In separating out the formation of the Z matrix from the main iteration sequence we have made the choice to opt for storage efficiency, rather than execution speed. This means that we have to re-broadcast the W_k vectors in order to do the $U = ZW$ matrix-vector product, and then perform a gathering operation to put a copy of the resultant vector U on every processor. The outerproduct UW^T can then be done locally. In the following code description remember that A stores both the W_k vectors and the Z matrix under construction.

```
FORMZ(A) –Inputs A_{N-2} and outputs Z in the A matrix
    Make the bottom right hand corner of A a 2 × 2 unit matrix
    FOR(K=N-2;K>0;K–) {
        IF(PROCNOFORROW[K] == MYPROCNO){
            recover W
             MPI_BCAST(W)
        }
        ELSE
            MPI_BCAST(W)
        }
        Do the local matrix vector multiplication U_l = AW
        MPI_ALLGATHER(U_l,U)
        Z = Z − 2UW^T
        Zero row k and column k of Z
        Set Z[k][k]=1
    }
```

3 Test Cases

The algorithms were tested using matrices that occur in many-body quantum mechanical model of crystals for which there are two states per atom. The total number of available states for the system is thus 2^N for a N atom system. The nature of the problem is such that the matrices for $l = 0, 1, \ldots, N$ particles on N crystal sites can be generated separately and the resulting matrices have dimension $C_l^N = \frac{N!}{(N-l)!l!}$. The general sparsity pattern of the test matrices is shown in figure 1 in which non-zero elements (about 3%) are all one and we store the original matrix as a dense matrix since the storage is needed for the eigenvectors anyway. It must be stressed too, that sparsity does not affect the complexity of the Householder algorithm and the complexity of the QL algorithm is dominated by the number of iterations required which is about $1.7N$ for the matrices we use although the maximum number of iterations required (to find the first eigenvalue) can vary markedly from 10 or so to something like $\sqrt{N}$.

The fact that the algorithm only requires minimal storage, for the original matrix only, is an important property when dealing with matrices that grow exponentially as the physical size of the problem is increased linearly. For instance for the 3432×3432 diagonalised in the scaling experiments requires about 100Mbytes of storage for the original matrix.

4 Results

The results presented below were all carried out on an IBM SP2 which is a heterogeneous 23 node machine with a subset of 16 'thin1' nodes on which our code was tested. Each 'thin1' node consists of a 66Mhz Power2 processor, 128Mbytes of RAM with a 64 bit memory bus, a 32Kbyte instruction cache and a 64Kbyte data cache. The latency and bandwidth for the machine, as measured by simple processor to processor 'ping-pong' benchmarks are 100μ seconds and 20 Mbytes/second[7].

For the scaling test presented in figure 2 the problem size was increased from $N = 70$ to $N = 3432$ with the number of processors fixed at 16. Ideally the

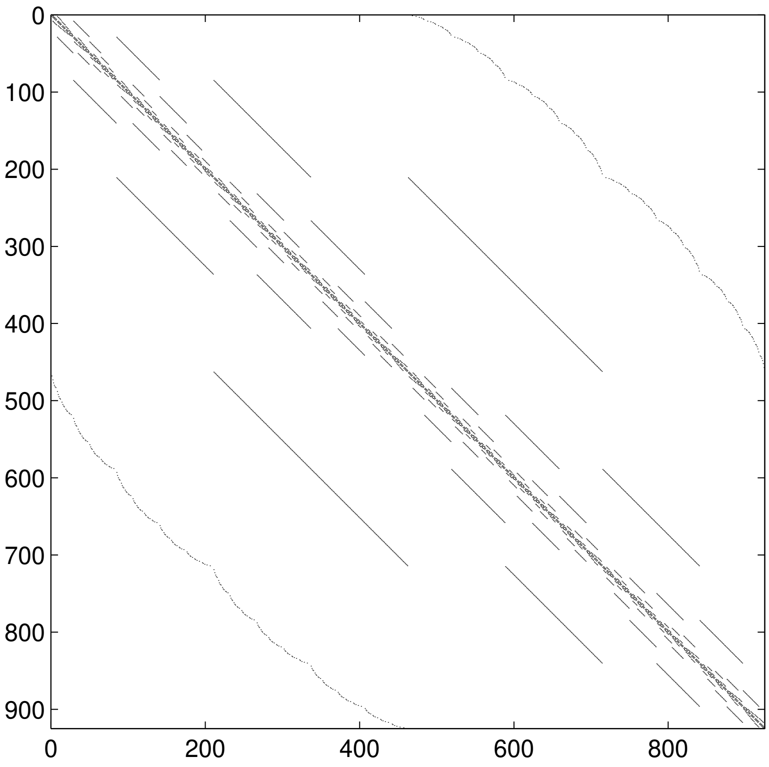

Figure 1: The Sparsity Pattern of the 924×924 Test Matrix used in the Speed-up Experiments.

P	1	2	3	4	5	6	7	8
HH	114	66.4	46.3	36.8	30.5	26.5	23.8	22.8
QL	37.1	28.2	21.7	17.8	15.4	13.5	12.1	11.3
Total	151	94.6	68.0	54.6	45.9	40.0	35.9	34.1
P	9	10	11	12	13	14	15	16
HH	21.0	19.9	18.8	18.6	17.5	17.0	16.9	16.9
QL	10.3	9.86	9.27	8.85	8.58	8.13	8.08	7.67
Total	31.3	29.8	28.1	27.5	26.1	25.1	25.0	24.6

Table 1: Execution Times (in seconds) on P Processors for the QL and Householder Segments of the Diagonalisation Algorithm for a 924×924 Matrix

N	70	126	252	462	924	1716	3432
HH	0.22	0.36	1.05	3.13	16.9	79.1	550
QL (Av. Iters.)	0.05 (1.71)	0.09 (1.67)	0.33 (1.71)	1.42 (1.75)	7.67 (1.89)	38.9 (1.75)	264 (1.89)
Total Time	0.27	0.45	1.38	4.55	24.6	118	814

Table 2: Execution Times (in seconds) on 16 Processors for the QL and Householder Segments of the Diagonalisation Algorithm for varying Problem Size, N. The Average number of iterations per eigenvalue for the QL algorithm is also indicated.

execution time should behave like $t = \alpha N^3$ for large N, which is indeed the case. The constant of proportionality is approximately 1.88 times larger than ideal as is seen from the overall efficiency of the algorithm on 16 processors.

The speed-up of the algorithm was tested on a problem size $N = 924$ on one to sixteen processors. The results presented on figure 3 show a difference between the efficiency(on 16 procesors). To estimate the efficiency we calculate the ratio of the time on one processor to P times the time on P processors. We find that the efficiency $E_{HH} \approx 0.50$, for the Householder routine and that of the QL algorithm is $E_{QL} \approx 0.46$. The super efficiency effect for the QL algorithm being due to a marked reduction in data cache misses when the data is distributed over a large number of processors. The overall efficiency $E \approx 0.49$ for 16 processors.

The measured data are shown in tables 1 and 2.

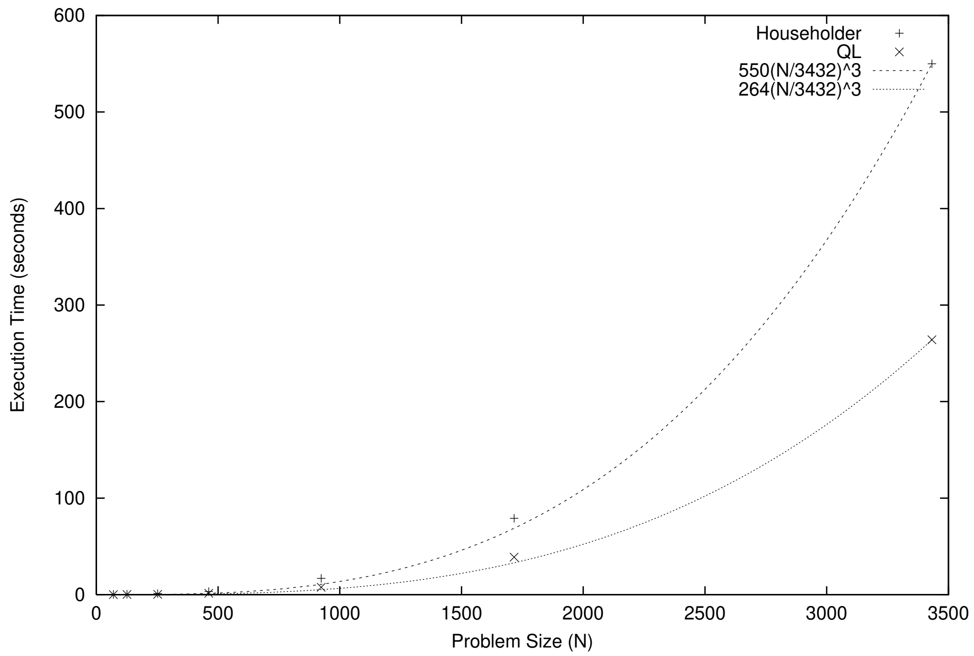

Figure 2: Execution Times for Varying Problem Sizes on 16 Processors

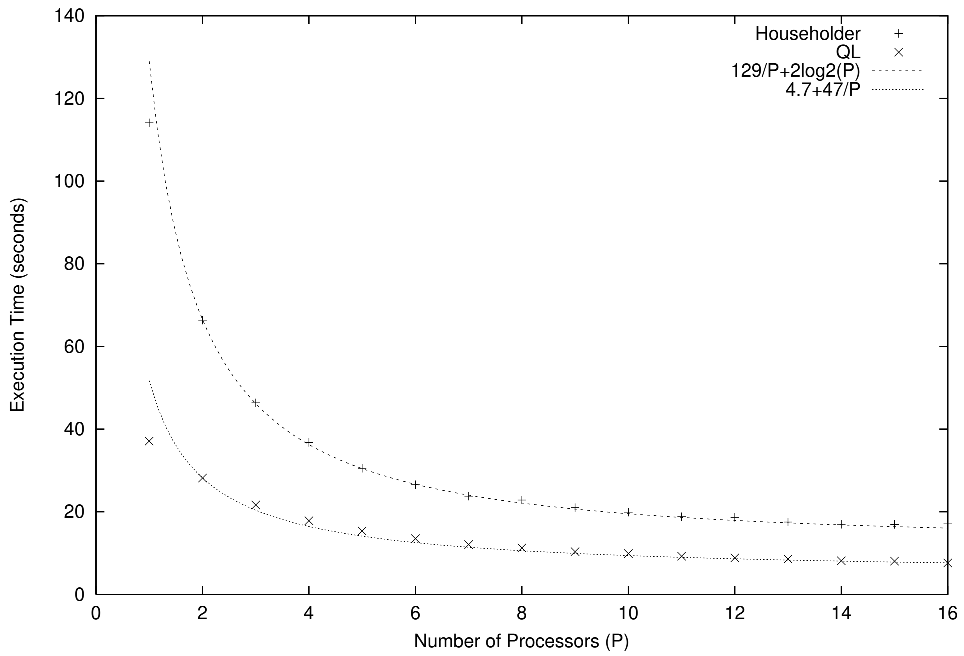

Figure 3: Execution Times for a 924×924 Matrix on 2 to 16 processors.

5 Summary

We have shown that a relatively straight forward parallelisation of the Householder tri-diagonalisation algorithm and the QL method for finding the eigenvalues and eigenvectors of the resulting matrix shows a time complexity like $\alpha N^3/P + \beta N^2 \log_2(P)$ for the Householder problem and like $\gamma N^2 + \delta N^3/P$ for the QL algorithm for fixed problem size on increasing numbers of processors, P. The $\log_2(P)$ term is empirically fitted to the data, but is consistent with *MPI_Bcast* and *MPI_Allgather* being implemented on a breadth first spanning out tree. If we use the timing results from figure 2 for large N, then we find values of $\alpha \approx \frac{8800}{(3432)^3}$, $\beta \approx 0.6 \times 10^{-6}$, $\gamma \approx 5.5 \times 10^{-6}$ and $\delta \approx \frac{4224}{(3432)^3}$. For the Householder algorithm, the communications penalty , $\beta N^2 \log_2(P)$, is less relevant for larger problems since the time for the main body of the algorithm is $\propto N^3$. If the eigenvalues are not required the QL algorithm remains sequential and the Householder method goes twice as fast. For large problems it might still worth using a parallel version of Householders algorithm since this spreads the memory requirements over all of the processors and there is still a 30% time saving.

References

[1] M.M. Chawla and D.J. Evans. *Parallel Householder Method for Linear-Systems* International Journal of Computer Mathematics 58 (1995) 159-167

[2] E.R. Davidson. *The Iterative Calculation of a Few of the Lowest Eigenvalues and Corresponding Eigenvectors of Large Real-Symmetric Matrices.* J. Computational Physics 17 (1975) 87

[3] F.J. Ferreira, P.B. Vasconelos and F.D. Dalmeida. *Performance of a QR Algorithm Inplementation on a Multicluster of Transputers* Computer Systems in Engineering 6 (1995) 363-367

[4] G. Fox et al, *Solving Problems on Concurrent Processors* (Prentice-Hall International Englewood Cliffs 1988).

[5] V. Kumar, A. Grama, A. Gupta and G. Karypis, *Introduction to Parallel Computing - Design and Analysis of Algorithms* (Benjamin Cummings California).

[6] J.H. Mathews, *Numerical Methods for Mathematics, Science and Engineering* (Prentice-Hall International Englewood Cliffs 1992).

[7] P. Melas and E.J. Zaluska, *High Performance Protocols for Clusters of Commodity Workstations* Lecture Notes in Computer Science 1470 (1998) 570-577.

[8] G.G. Meyer and M. Pascale *A Family of Parallel QR Factorisation Algorithms* Concurrency-Practice and Experience 8 (1996) 461-473

[9] B.N.Parlett, H.Simon and L.M.Stringer *On Estimating the Largest Eigenvalue With the Lanczos Algorithm* Mathematics of Computation 38 (1982) 153-165

[10] W.H.Press, B.P. Flanery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipies - The Art of Scientific Computing* (Combridge University Press Cambridge 1986).

[11] W.H.Press, B.P. Flanery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipies in Fortran 90- The Art of Scientific Computing* (Cambridge University Press Cambridge 1996).

[12] M.A. Shalaby. *A Parallel Processed Scheme for the Eigenproblem of Positive-Definite Matrices* J. of Computational and Applied Maths. 54 (1994) 99-106