Parallel Application Software on High Performance Computers

Parallel Diagonalisation Routines.

The CCLRC HPCI Centre at Daresbury Laboratory

R.J. Allan and I.J.Bush

Edition 2: 22nd August 1996

Abstract

In this report we list diagonalisation routines available for parallel computers. The methodology of each routine is outlined together with benchmark results on a typical matrix where available. Storage requirements and advantages and disadvantages of the method are also compared. The vast majority of these routines are available for real dense symmetric matrices only, although there is a known requirement for other data types – such as Hermitian or structured sparse matrices. We will report on new codes as they become available.

This report is available from http://www.dl.ac.uk/TCSC/HPCI/

Contents

1 Summary

Many of the computer codes under development within the HPCI Grand Challenge Consortia involve matrix diagonalisation as the rate determining step, and there is a pressing need for access to efficient parallel implementations, particularly for symmetric full matrix algorithms. A number of these have thus been tested in collaboration with other Consortia under the common management of the Daresbury Centre. Some codes for sparse matrices are also discussed.

For general information about eigenproblems we refer the reader to Numerical Recipes [57], Golub and van Loan [32], Freeman and Philips [31], Isaacson and Keller [38], Dahlquist et al. [18], Parlett [55], Wilkinson [61] or other similar texts.

The algorithms broadly fall into three camps:

1. Householder reduction to tridiagonal form followed by QL or QR diagonalisation;
2. iterative procedures, either one or two-sided Jacobi;
3. iterative procedures based on subspace decomposition, conjugate gradient, Lanczos and Arnoldi methods.

Known parallel implementations are:

- Cleve Moler and Richard Chamberlain's EISCUBE released by Intel
- ScaLAPACK PDSYEVX (included in Cray and IBM libraries)
- PNL PDSPEV in PeIGS2.1 package
- Parallel eigensolver by Hendrickson, Smith, Jessup known as the HJS algorithm.
- Ian Bush's one-sided block Jacobi BFG
- Pittsburgh Supercomputer Center 2-sided Jacobi PJAC (can also do complex symmetric or Hermitian)
- Parallel NAG library
- Kristi Maschhoff's parallel ARPACK (can handle large sparse and unsymmetric matrices)
- PRISM project SYISDA invariant subspace decomposition approach
- Chris Potter's parallel divide and conquer algorithm
- Soren Paedkjar's Spectral Transform Lanchos Method

Some notes on the methodology and critique as to their application on distributed-memory parallel computers are given below together with benchmark speedup results for comparison purposes.

1.1 Test Results

The test data was generated using a program written by Alistair Rendell which assembles a pseudo Fock matrix relevant to computational chemistry problems. The matrix type chosen is diagonally dominant, seeded with uniform random numbers, although other types could be tested. The symmetric matrix is initialised using SLARNV() or equivalent and diagonal elements are multiplied by 1000 to introduce diagonal dominance. This data was used for both the Jacobi and Householder algorithms, although clearly the Jacobi method, because it is iterative, becomes better the closer the matrix is to diagonal.

1.2 Recommendations

In Figure 1 we compare the execution times for representative routines for an order 1024 matrix. The figure is based on results from the T3D. It may be seen that whilst on small numbers of nodes the Jacobi methods are typically three to four times slower than those based on QL factorisation (a similar ratio to that found in serial environments), the Jacobi methods scale better, to the extent that on 256 processors the BFG method is the fastest. We find that this is because the QL factorisation does not scale very well. The reason for the poorer scaling of PDSPEV compared to EISCUBE or ScaLAPACK PDSYEVX is the Gram-Schmidt orthogonalisation, which, being essentially a QL step, scales poorly. Our remarks however are largely applicable to any of the direct methods as compared to the Jacobi iterative methods.

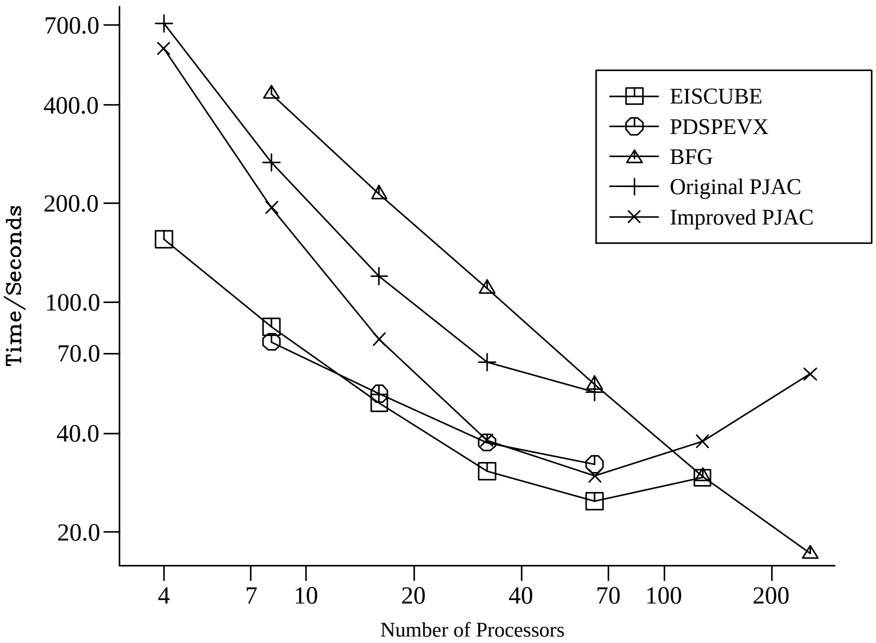

Figure 1: Variation of Time to Solve Matrix of N=1024

The one-sided Jacobi method scales better than the two-sided. Although we have not investigated this in detail, we suspect that it is because the grain size in the one-sided method remains essentially constant with increasing numbers of processors, while for the two sided it steadily decreases.

Thus the observed execution time is a result of a competition between the algorithm's serial speed and it's scalability. The fast serial routines, based upon Householder reduction, show the poorest scaling. On the other hand the slowest serial method, one sided Jacobi, scales more linearly for the above example (14.6 speed up on going from eight to 128 processors, versus 2.9 for EISCUBE). Two-sided Jacobi shows moderate serial speed, and moderate scalability. We might expect, therefore, that each of the methods has a regime where it is the quickest, and this is illustrated as a 'phase diagram' in Figure 2. The vertical axis on this figure is a measure of the amount of data on each processor, large values implying small processor numbers.

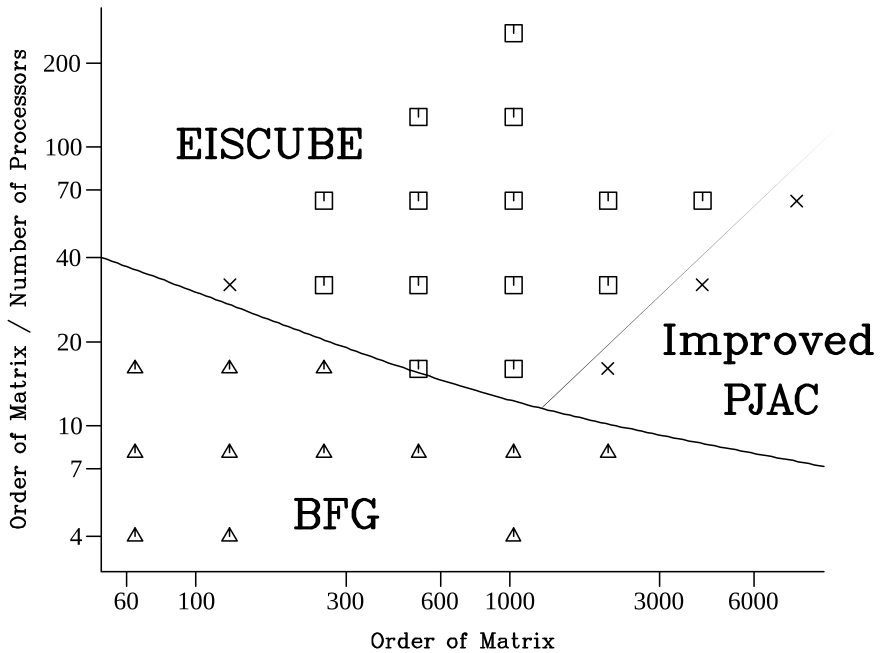

Figure 2: Phase Diagram Showing Region of Interest for Diagonalisers Tested

It can be seen that the above discussion of the scaling behaviour of the algorithms is borne out. Large processor counts favour the best scaling algorithm, BFG. Small counts favour the best serial algorithm. For very large matrices on large numbers of processors the reasonable scaling and serial speed of the PJAC routine makes it the fastest. One might expect similar arguments to extend to the class of conjugate gradient or subspace decomposition methods which scale through their dependence on matrix multiplication. We have not carried out a detailed analysis however as current implementations are less well developed than those in the other classes.

On raw speed grounds we therefore make the following recommendations:

- For an order-n matrix on P processors, $N/P < 8$, one-sided Jacobi will be the best
- For matrices roughly less than 3000×3000 and N/P greater than 8, direct methods win
- For matrices of order greater than about 3000 and N/P greater than 8, two-sided Jacobi is fastest.

Among the direct methods on systems with low latency communications (e.g. Cray T3D, see section 7) block cyclic distribution as in ScaLAPACK using BLACS is an advantage. However on other systems (e.g. IBM SP2) EISPACK or PeIGS will win.

There are other factors that should be considered, however, not least accuracy and stability. The failure of EISCUBE to ensure orthonormality of the eigenvectors is its biggest failing, and may cause PDSPEV, although slightly slower, to be favoured in practical circumstances. The ScaLAPACK routine PDSYEVX, **whilst not guaranteeing orthogonality**, has been found to yield orthogonal eigenvectors in most situations, depending on the amout of workspace provided, and information is available to the user to test that this is true. Our experience further suggests that Jacobi methods tend to be more stable than Householder. Another important factor is flexibility. The PeIGS library certainly provides the most flexible user interface, solving both the normal and general eigenproblem, allowing the user to specify exactly the data distribution involved, and coping with packed symmetric matrices. The ScaLAPACK and PARPACK routines use data distribution relevant to the BLACS and PBLAS which are also very flexible if block-cyclic formats are suitable. Of the other routines BFG allows some flexibility in the data distribution. PJAC has the further deficiency of not being able to solve an arbitrary-sized problem.

The use of memory for workspace is highly variable in these methods. Clearly the iterative methods are able to work in situ, but other methods tend to need at least memory equivalent to one extra matrix. Unfortunately it is difficult to quantify this further as much depends on data distribution, blocking factors etc. However in practice there may be a trade-off between available memory and cpu efficiency for large problems.

We should finally note that we have considered only one type of matrix in this report: less strongly diagonally-dominant matrices will favour direct methods, while matrices that are more diagonally dominant will be better treated by iterative methods. However we feel that the basic structure of the phase diagram will remain unchanged, only the exact position of the transition lines will move.

2 Direct methods

The traditional method for determining the eigensystem of a real, dense symmetric matrix A employs a three-step technique [32]. First A is reduced to tridiagonal form using a series of Householder transformations (reflections). Next the eigensystem of the tridiagonal matrix T is computed. The eigenvalues of A are the same as those of T, while the eigenvectors of A are found in the third step by backtransforming the eigenvectors of T via reduction transformations (see e.g. [59])

Although the sequential algorithms for the tridiagonalisation and backtransformation steps are well understood, tridiagonal eigensolvers continue to be an area of active research. With respect to parallel algorithms the efficiency criteria and therefore the situation is somewhat reversed. There has been much work on the tridiagonal eigensolver since Huang's 1974 study of multisection methods on the ILLIAC IV [37]. This continued with the implementation of several other tridiagonal eigensolvers on a variety of shared and distributed-memory systems [7, 8, 9, 10, 23, 43, 49, 48, 50].

There is still however the problem of assuring orthogonality which is not fully worked out. Most commonly the parallel implementations use bisection to find the eigenvalues, followed by inverse iteration to get the vectors. However this does not guarantee orthogonality if the values are clustered. In this case a modified Gram-Schmidt procedure is needed to reorthogonalise. However if clustered values belong to different processors this involves extra communication [29, 43].

Work on parallel tridiagonalisation and backtransformation began more recently and was confined to distributed-memory systems. Chang et al. described, but did not implement two parallel versions of the Housholder algorithm [12, 13]. Several implementations of the traditional tridiagonalisation algorithm have appeared recently [24, 48, 34, 15]. A different method was described by Hegland [33]. Subspace decomposition methods are described in a separate section below.

Whilst the QL method is often the method of choice on sequential machines when the system is not very diagonally dominant, the QL and QR methods are generally considered to give problems on a parallel computer because of poor scaling. There are several reasons for this. Most time spent in the method is in the Householder reduction to tridiagonal form, this may be up to 95% of the time.

However the QL and QR methods produce eigenvalues which are not ordered. This means that closely similar (or even degenerate to machine precision) values may be on different processors. We cannot guarantee orthogonality of eigenvectors from similar eigenvalues if they are computed on different processors, as usually eigenvectors are orthogonal to the previously found one on the same processor. In general an explicit parallel Gram-Schmidt orthogonalisation step would be required. This could result in poor load balancing in the full algorithm. No orthogonalisation is done in EISCUBE or ScaLAPACK, which therefore **do not guarantee to give orthogonal eigenvectors**.

In the PeIGS software repeated inverse iteration with orthogonalisation is used. Implicit orthogonalisation as part of the inverse iteration can however be a large overhead in cases where the eigenvalue spectrum does not have evenly spaced values. The HJS code does an explicit Gram-Schmid step.

We now discuss the various implementations of the traditional methods that we have found.

2.1 EISCUBE QR

The oldest parallel Householder-QR method implemented by Cleve Moler and Richard Chamberlain of Intel SCD is based on EISPACK routines and uses Sturm sequences with bisection and perfect-shift QR iterations to find the eigenvalues and vectors respectively.

The input matrix is first reduced to tridiagonal form using Householder reflections to zero out successive columns below the principal sub-diagonal. This reduction does not parallelise perfectly, since, although transformations can be applied in parallel to the remainder of the matrix, the transformations must be determined in sequence and broadcast. This results in communication overheads and potentially poor load balancing for $p \sim$n.

The characteristic polynomial $p_n(\lambda)$ of the tridiagonal matrix can be evaluated for any trial value of λ by recursion. The polynomials obtained during recurrence form a Sturm sequence which is used to localise the eigenvalues within intervals on the real axis. Bisection is then used to refine the intervals and find the eigenvalues in parallel.

Eigenvectors are found using perfect shifts and QR iterations, also in parallel.

A possible alternative is to first obtain the eigenvalues using implicit QR or QL iteration and then use inverse iteration to find the eigenvectors.

The routines require data stored by columns (or rows). The column i is stored on processor mod(i,p) where p is the total number of processors. There must be at least one column per processor, which limits the application to $p \leq$n.

Eigenvectors are **not guaranteed to be orthogonal** in EISCUBE.

Table 1: Performance of Routine EISCUBE on the Cray T3D

Order of	Number of Nodes [all times in seconds]						
Matrix	4	8	16	32	64	128	256
8	0.023	0.044					
16	0.029	0.044	0.175				
32	0.050	0.076	0.119	0.347			
64	0.139	0.147	0.203	0.272			
128	0.618	0.449	0.480	0.496			
256	3.302	2.109	1.592	1.490			
512	21.04	12.18	7.849	5.681	5.9218		
1024	155.7	84.44	49.64	30.637	24.879	29.26	
2048		876.5		244.82	156.86	140.39	225.26
4096					1119.7	770.55	930.40
8192						5027.5	

2.2 ScaLAPACK

The ScaLAPACK (or Scalable LAPACK) library includes a subset of LAPACK (Linear Algebra PACKage) routines [4] redesigned for distributed memory MIMD parallel computers [14]. It is rapidly becoming the default library on MPP systems e.g. Cray T3D and IBM SP2 (see libsci or PESSL documentation [41]). ScaLAPACK is therefore portable across the Cray T3D, Intel series, IBM SP series, Thinking Machines CM-5, and clusters of workstations.

Like LAPACK, the ScaLAPACK routines are based on block-partitioned algorithms. In the ScaLAPACK routines, all interprocessor communication occurs within the PBLAS (usually linked to optimised architecture dependent BLAS) and the BLACS (linked to architecture dependent communication primitives). These use a block-cyclic data distribution and are further discussed below. It is important to note the comments on the Cray T3D implementation!

The first public release (version 1.0) of the ScaLAPACK software library occurred on February 28, 1995. There are more arguments on ScaLAPACK routines than in LAPACK because distributed matrices need an additional descriptor array and a context is used to specify the underlying communication topology. The goal is that, apart from such novel features, ScaLAPACK routines resemble their LAPACK equivalents as much as possible.

The list of routines is still growing. Information is available from the Web via:

http://www.netlib.org/scalapack/index.html .

There is currently no Users' Manual, but LAWNs (LAPACK Working Notes) contain much of the relevant information. see:

http://www.netlib.org/lapack/lawns .

The main categories of routines from ScaLAPACK version 3 release is as follows:

- subset of LAPACK for distributed memory machines
- LU, Cholsky and QR factorisation
- reduction to tridiagonal, Hessenberg and bidiagonal forms
- symmetric eigenvalue problems (version 2.0 also includes Hermitian and complex forms)

2.2.1 BLACS and PBLAS

The fundamental building blocks of the ScaLAPACK library (and also PARPACK, below) are parallel distributed memory versions (PBLAS) of the Level 1, 2 and 3 BLAS, and the BLACS (Basic Linear Algebra Communication Subprograms) for communication tasks that arise frequently in parallel linear algebra computations.

These were designed assuming that matrices are tiled, each tile distributed in a two-dimensional block cyclic decomposition. Blocks of size d×d are mapped to individual processors (in principle could be rectangular but in practice not yet). A range of actual data distributions can be obtained in practice, since tiles are mapped onto a rectangular processor template. If there are k×m processors the data in a tile is of total size kd×md. Row or column-major processor templates are possible. In practice optimal performance is realised only if the tiles are nearly square. The dimension of the whole matrix n must be a multiple of d, MOD(d,n)==0.

It is important to note that on the Cray T3D the BLACS and therefore the PBLAS are implemented directly using the SHMEM communication primitives. This requires that all data communicated is statically declared through the use of Fortran COMMON blocks. This statement therefore applies to both the original matrices and all workspace used in the ScaLAPACK routines, and it is not generally possible to dynamically allocate space since spurious error messages may arise on the T3D when the SHMEM routines are called from the BLACS.

No such problem exists on other platforms where usually MPI ot PVM is used as the underlying communication layer.

The PBLAS required extensive modifications to run on the T3D. The interface was however preserved – except for one change. Users of ScaLAPACK on the T3D will be required to initialise an internal buffer prior to calling PBLAS or higher level ScaLAPACK routines. This is done by calling a new external routine `initbuff`. One initialisation is required for "unlimited" access to the ScaLAPACK routines. Please refer to the documentation on *http://patpwww.epfl.ch/scalapack/src/man/initbuff.html* .

The Cray specific subroutine `initbuff` must be called to initialise a statically declared workspace used by the PBLAS. This is described in the MPP man pages. Specifically for the PSSYEVX routine a the workspace must be of size length (in bytes):

length=4*maxb*(((maxd//minb)/minp)*maxb+maxb)*8

where

```
minb      Minimum block size in the distribution of any array; for
          example, if the routine requires two arrays, and if these
          are block-cyclically distributed using 3-by-4 and 4-by-2
          blocks, minb should be taken to be 2.

maxb      Maximum block size in the distribution of any array; in the
          preceding example, maxb = 4.

minp      Smaller side of the two-dimensional grid over which the
          arrays are distributed; for example, if the grid is 4-by-2,
          minp=2.

maxd      Maximum length in any (local) dimension of any array passed;
          for example, if the routine requires two arrays, and if
          these are l-by-k and k-by-m then maxd=MAX(l,k,m).
```

2.2.2 Routine PDSYEVX

PDSYEVX computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix A by calling the recommended sequence of ScaLAPACK routines. Eigenvalues/vectors can be selected by specifying a range of values or a range of indices for the desired eigenvalues. A variant of this routine is also available in the PeIGS2.1 software (see below). Single precision and complex versions also exist in the ScaLAPACK library. This routine is included in the library PESSLP2 for IBM PS2 systems and the equivalent PSSYEVX is included in the default library libsci.a on Cray T3D systems.

A description vector is associated with each two-dimensional (2D) block-cyclically distributed matrix. This vector stores the information required to establish the mapping between a matrix entry and its corresponding process and memory location. Note that the vector should not be accessed directly as its definition is changing. A routine `descinit` is provided to initialise it. (Unfortunately this routine is not yet present in the IBM library).

Note non-square blocking factors are currently not permitted in PDSYEVX.

Either the upper or lower triangle of the input matrix can be defined. This is used as workspace by the routine, and therefore input data is lost. The amount of scratch memory used for this routine is approximately $5n+5n^2/p$ double precision words if all eigenvectors are required. In the PESSLP2 version it is sufficient to call PDSYEVX with lwork=0, liwork=0 and it will automatically allocate the required workspace.

The workspace which must be provided depends on many factors, including block size, number of processors and number of eigenvectors requested. We found $5n+5n^2/p$ words to be sufficient for all eigenvectors with the largest possible square blocks. A more exact estimate, is given in the following subroutine according to the PESSL reference manual.

```
SUBROUTINE worksize(n,nb,nprow,npcol,lwork,liwork)
IMPLICIT NONE
INTEGER n,nb,lwork,nprow,npcol,liwork
```

```
      INTEGER nn,neig,np0,nq0,isizestein,isizestebz
      EXTERNAL numroc,iceil
      INTEGER numroc,iceil
      nn=MAX(n,nb,2)
      neig=n
      np0=numroc(nn,nb,0,0,nprow)
      nq0=MAX(numroc(neig,nb,0,0,npcol),nb)
      lwork=5*n+MAX(5*nn,np0*np0)+iceil(neig,nprow*npcol)*nn+2*nb*nb
      isizestein=3*n+nprow*npcol+1
      isizestebz=MAX(4*n,14)
      liwork=MAX(isizestein,isizestebz)+2*n
      WRITE(6,'('' workspace required '',2i10)')lwork,liwork
      END SUBROUTINE worksize
```

A plateau of minimum memory requirement is reached for nb<500. We shall return to the question of defining nb below.

As has already been stated several times, orthogonalisation of the eigenvectors can be a problem on parallel computers. PDSYEVX does not promise orthogonality for eigenvectors that are associated with tightly clustered eigenvalues. PDSYEVX does not re-orthogonalise eigenvectors that are on different processors. The extent of re-orthogonalisation is controlled by the amout of workspace available, and above represents a minimum.

Some care has however been taken in the implementation of PDSYEVX to aid the user in this respect. If the work space provided is too small to guarantee orthogonality, PDSYEVX attempts to maintain orthogonality in the clusters with the smallest spacing between the eigenvalues. If the workspace is large enough vectors are re-orthogonalised if eigenvalues are closer than a specified limit, usually around 10^{-3}. In the event of eigenvectors not being re-orthogonalised information is output in some new dummy arguments which specify which eigenvector has not been treated and how close its eigenvalue is to other ones in the same cluster. The user can then make a decision based on this information, possibly invoking an external parallel Gram-Schmidt procedure. Note that this is built into the otherwise similar PNL software PeIGS. Automatic routines are promised in future releases of ScaLAPACK.

2.2.3 Sample calling program

```
      PROGRAM sample_pdsyevx_call
***********************************************************************
*  -- ScaLAPACK example routine (version 1.0)
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     December, 1994.
*
*     This routine contains a sample call to PDSYEVX.
*     Written by Ken Stanley, modified by RJA
***********************************************************************
* number of procs expected
```

```
      INTEGER,  PARAMETER :: MP=8
* problem size
      INTEGER, PARAMETER :: n=512
* subblock size
      INTEGER nbx,nby,nb
* number of process rows and cols
* and size of matrix required per processor
      INTEGER nprow,npcol,maxy,maxx,ier
      PARAMETER (nprow=4,npcol=2,maxx=n/npcol,maxy=n/nprow)
* workspace, blaswork approx proportional n*n (guessed here)
      INTEGER LWORK,LIWORK
      DOUBLE PRECISION MONE,zero
      INTEGER CONTEXT,I,IAM,INFO,M,,NPROCS,NZ
      INTEGER DESCA(9),DESCZ(9),ICLUSTR(MP*2),IFAIL(N)
      DOUBLE PRECISION   A(Maxy,Maxx),GAP(MP),W(N),Z(Maxy,Maxx)
      REAL*8, DIMENSION(:), ALLOCATABLE ::  work
      INTEGER, DIMENSION(:), ALLOCATABLE ::  iwork
#IFDEF Cray-t3d
* space required for internal PBLAS buffer
      INTEGER, PARAMETER :: blaswork=100000
      INTEGER pblasbuf(blaswork)
#ENDIF
      EXTERNAL BLACS_EXIT,BLACS_GET,BLACS_GRIDEXIT,
     &         BLACS_GRIDINFO,BLACS_GRIDINIT,BLACS_PINFO,
     &         BLACS_SETUP,PDLAMODHILB,PSSYEVX
* common blocks to ensure static allocation is preserved for SHMEM
      COMMON/shmemblk/a,gap,w,work,z,desca,descz,iclustr,pblasbuf
      DATA MONE/-1.0D0/,ZERO/0.0D0/
* define subblock size, optimal c.10-30
      nb=32
* Initialize the BLACS
      CALL blacs_pinfo(iam,nprocs)
      IF(iam.NE.0)OPEN(6,FILE='/dev/null',STATUS='unknown')
      WRITE(6,'('' running on nprocs = '',i4)')nprocs
* Initialize a single BLACS context
      CALL blacs_get(-1,0,context)
      CALL blacs_gridinit(context,'r',nprow,npcol)
* allocate minimum work space
      CALL worksize(n,nb,nprow,npcol,ilwork,iliwork)
      ALLOCATE(work(lwork),STAT=ier)
      ALLOCATE(iwork(liwork),STAT=ier)
* These are basic array descriptors
      CALL descinit(desca,n,n,nb,nb,0,0,context,maxy,info)
* same for eigenvectors
      CALL descinit(descz,n,n,nb,nb,0,0,context,maxy,info)
*  Build a matrix that you can create with
*  a one line matlab command:  hilb(n) + diag([1:-1/n:1/n])
      CALL pdlamodhilb(n,a,1,1,desca,info)
```

```
*  uncomment this line to see the matrix printed out.
*   CALL pdlaprnt(n,n,a,1,1,desca,0,0,'a',6,work)
*  Ask PDSYEVX to compute the entire eigendecomposition
#IFDEF Cray-t3d
* see comments below
      needs=4*nb*(n/MIN(nprow,npcol)+nb)
      WRITE(6,'('' needs '',i8,'' blaswork '',i8)')needs,blaswork
      CALL initbuff(pblasbuf,needs*8)
      CALL pssyevx('v','a','u',n,a,1,1,desca,zero,zero,13,
     &              -13,mone,m,nz,w,mone,z,1,1,descz,work,
     &              lwork,iwork,liwork,ifail,iclustr,gap,info)
* Print out the eigenvectors
      CALL pslaprnt(n,n,z,1,1,descz,0,0,'z',6,work)
#ELSE
      CALL pdsyevx('v','a','u',n,a,1,1,desca,zero,zero,13,
     &              -13,mone,m,nz,w,mone,z,1,1,descz,work,
     &              lwork,iwork,liwork,ifail,iclustr,gap,info)
      CALL pdlaprnt(n,n,z,1,1,descz,0,0,'z',6,work)
#ENDIF
* Print out matlab code which will check the residual
      IF(iam.EQ.0)THEN
         PRINT *,' N =',N
         PRINT *,' A= hilb(N) + diag([1:-1/N:1/N])'
         DO I=1,N
            PRINT *,' W(',I,')=',W(I),';'
         END DO
         PRINT *,' ortho=A - Z * diag(W) * Z'' '
         PRINT *,' resid=A * Z - Z * diag(W)'
         PRINT *,' norm(ortho)'
         PRINT *,' norm(resid)'
      END IF
      CALL blacs_gridexit(context)
      CALL blacs_exit(0)
* Uncomment this line on SUN systems to avoid the useless print out
*      CALL ieee_flags('clear','exception','underflow','')
 9999 FORMAT('W=diag([',4D16.12,']);')
      END
*
      SUBROUTINE pdlamodhilb(n,a,ia,ja,desca,info)
*********************************************************************
* -- ScaLAPACK example routine (version 1.0)
*  Univ. of Tennessee,Univ. of California Berkeley,NAG Ltd.,
*  Courant Institute,Argonne National Lab,and Rice University
*  December,1994
*********************************************************************
      DOUBLE PRECISION   ONE
      PARAMETER          (ONE=1.0D0)
      INTEGER            IA,INFO,JA,N
```

```
      INTEGER            DESCA(8)
      DOUBLE PRECISION   A(*),temp
      INTEGER            I,J,MYCOL,MYROW,NPCOL,NPROW
      EXTERNAL           BLACS_GRIDINFO,PDELSET
      INTRINSIC          REAL
      INFO=0
      CALL blacs_gridinfo(desca(7),nprow,npcol,myrow,mycol)
      IF(IA.NE.1)THEN
         info=-3
      ELSE IF(JA.NE.1)THEN
         info=-4
      END IF
* sample data, make it diagonally dominant
* all operations in here are local
      DO J=1,N
         DO I=1,N
            CALL infog2l(i,j,desca,nprow,npcol,myrow,mycol,iia,jja,
     &           iarow,iacol)
* skip if its not for me
            IF(myrow.ne.iarow.or.mycol.ne.iacol)CYCLE
            temp=one/REAL(i+j)
            IF(i.eq.j)temp=temp+100.0D0*REAL(N-I+10)/REAL(N)
            a(iia+(jja-1)*desca(8))=temp
         END DO
      END DO
      END
```

Not all routines are present in the IBM PESSL library [41]. In particular the descinit routine is missing and the user must load the desc vector with its 8 arguments in their given order (not recommended in ScaLAPACK versions).

2.2.4 Benchmark results

Both the amount of workspace required and execution time are strongly dependent on the block size nb. Since we found the workspace decreases almost monotonically as nb decreases for the matrix sizes tested, we now consider the effect on execution time. The following results were obtained on 4 processors of the IBM SP2 at Laboratoire SIMPA, Université de Rennes I, with matrix size approximately 2000.

The explosion in cpu time for the cases with large block size is likely to be due in the main to the system paging through the workspace array which rapidly increases in size. This is improved for the smaller blocks because both workspace and block size is smaller. The example worksize routine above computes the minimum size required. In the PESSL implementation setting the arguments to 0 allocates this automatically.

A minimum in execution time is seen for 10<nb<50. Within these limits the dependence is not strong.

Table 2: Variation of PDSYEVX performance with block size

Order of	Matrix size [times in seconds]		
nb	2000 2x2	2048 2x2	2000 1x4
1024		>1000	
512		486.3	
256		346.0	
200	345.7		
100	228.0		
64		236.4	
50	207.7		229.6
32		215.3	
25	200.7		232.1
20	200.2		211.8
16		229.4	
10	204.8		212.9
8		234.9	

The increase in time for nb<20 is likely to be due to overheads in starting the LAPACK and BLAS routines for each block.

It is assumed that this behaviour will apply equally to larger problem sizes, and a fixed block size of nb=32 was therefore used for benchmark runs on both IBM and Cray. For a particular application it may be worth investigating in detail the best choice of block size if the code permits sufficient flexibility.

Table 3: Performance of PSSYEVX from Cray T3D libsci.a

Order of	Number of Nodes [times in seconds]						
Matrix	4	8	16	32	64	128	256
	2x2	4x8	4x4	8x4	8x8	16x8	16x16
256	2.41	2.17					
512	8.68	6.82	5.04				
1024	51.4	29.0	17.9				
2048	299.0	144.0	76.7	52.3	35.2		
4096			410.4	243.7	176.6	110.3	
5120					233.3	169.7	
8192						491.3	

A sudden increase in execution time is also seen in this table for large matrices on small numbers of nodes. This is again due to the system paging and should be avoided in practice.

2.3 PNL Version of QL Routine PDSPEV

PeIGS [27] is a collection of now commonly used linear algebra subroutines for computing the eigensystem of the real standard problem $\mathbf{Ax} = \lambda\mathbf{x}$ and the general eigensystem $\mathbf{Ax} = \lambda\mathbf{Bx}$, with dense

Table 4: Performance of PDSYEVX from IBM SP2 AIXv4.1 PESSLP2

Order of	Number of Nodes [times in seconds]						
Matrix	1	2	3	4	8	16	32
	1x1	2x1	3x1	2x2	4x2	4x4	8x4
512	13.3	11.0		7.89			
768	32.9	24.7	20.9	16.7	13.8		
1024	203.7	83.3		39.1	24.8		
1536	642.7	240.2	148.7	103.2	54.9		
2048		534.4		215.3	104.4		
3072			972.5	626.7	278.3	155.4	
3200				701.3	342.5	170.0	
4096				1592	580.0	306.8	207.8
5120					1389	587.4	

real symmetric matrices and **B** positive definite.

It was developed by David Elwood, George Fann and Richard Littlefield of Batelle Pacific Northwest Laboratory, USA with whom the HPCI Centre at Daresbury Laboratory has a research collaboration. It is documented [27] and there are also papers benchmarking several methods tested at PNL [48, 28].

We consider the routine PDSPEV(), a parallel implementation equivalent to LAPACK routine DSPEV(). A routine for selecting eigenvalues, PDSPEVX, is also provided

The diagonalisation method proceeds by Householder reduction to tridiagonal form using a parallel version of the EISPACK routine TRED2(). Processor 0 broadcasts the diagonal and sub-diagonal of the tridiagonal matrix to all processors, which each compute a set of eigenvalues using DSTEBZ() from LAPACK. This uses Sturm sequencing and bisection, some redundant computation is performed if there are closely clustered eigenvalues, just by having the intervals overlapping slightly.

Processor 0 then collects all eigenvalues and block numbers and broadcasts them. Inverse iteration is used with a modified Gram-Schmidt procedure for re-orthogonalisation to find the **orthogonal** eigenvectors [29]. This gives results comparable to LAPACK routine DSTEIN().

Standard back-transformation with the Householder matrix gives the eigenvectors of the original problem.

PeIGS uses a flexible column distribution with packed storage for real symmetric matrices. The user specifies the processor which stores each column. Basic linear algebra operations, such as triangular matrix multiplication are performed in a panel-block systolic loop [20]. The program is capable of taking data in a column (row)-wise distribution with processor mapping specified in integer vectors. It can also handle data in compact lower triangular format, which is useful. This is done by pointers and dynamic memory allocation in the C routines. Scratch memory of about $4n^2/p$ is required. The authors are working to reduce this requirement where possile.

The driver routine for the general eigenvalue problem is PDSPGV(). The problem is reduced to a standard one by Cholesky factorisation of **B**, so $\mathbf{B} = \mathbf{L}\mathbf{L}^T$ where **L** is a lower triangular matrix. This exists providing **B** is symmetric and positive definite, which guarantees real square roots. Hence the conditions on using PDSPGV.

We have: $\mathbf{Cx} = \lambda\mathbf{x}$ where $\mathbf{C} = \mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-T}$.

The parallel routine CHOLESKI() uses a submatrix algorithm, and instead of doing two solves to form $\mathbf{C}$ the $\mathbf{L}$ is inverted using a routine INVERSEL() and a lower triangular and upper triangular matrix multiply are performed.

The back-transformation step must also include $\mathbf{L}^{-1}$ for the general case.

2.3.1 Sample calling program in F90

```
      PROGRAM test_pigs
      IMPLICIT NONE
      INCLUDE 'mpif.h'
      INTEGER, PARAMETER :: n=8192
      INTEGER rsize,ptrsize,isize
      INTEGER iproc,inode,nnode,ier
      INTEGER mapa(n),mapz(n)
      INTEGER i,j,k,ii,k2
      REAL*8 eval(n),start,finish,time
! assume square arrays for this example
      REAL*8, DIMENSION(:,:), ALLOCATABLE ::  matrixa
! maximum sizes of storage vectors required for packed lower-triangular
! format of symmetric arrays. Will only use approx
! n*(n+1)/(2*nnode) of this
      INTEGER meigval,info,ivector,index,irange,ilb,iub,ier
      REAL*8 abstol,lb,ub
      INTEGER, DIMENSION(:), ALLOCATABLE :: iscratch
      REAL*8, DIMENSION(:), ALLOCATABLE :: dptr,scratch
      REAL*8, DIMENSION(:), ALLOCATABLE :: veca,vecz
      REAL*8, EXTERNAL :: RTC
      call mpi_comm_rank(mpi_comm_world,inode,ier)
      call mpi_comm_size(mpi_comm_world,nnode,ier)
      WRITE(6,'('' inode, nnode = '',2i4)')inode,nnode
      ALLOCATE(matrixa(n,n/nnode),STAT=ier)
      ALLOCATE(veca(n*n/nnode),STAT=ier)
      ALLOCATE(vecz(n*n/nnode),STAT=ier)
! fill matrix with numbers here:
      matrixa=0.2d0
! set the column mapping of processors
      DO i=1,n
        iproc=MOD(i,nnode)
        mapA(i)=iproc
        mapZ(i)=iproc
      END DO
! now copy contents of array into a packed local vector according to
! this mapping
      k=0; k2=0
```

```
! for each row
      DO i=1,n
        IF(mapa(i)/=inode)CYCLE
        k2=k2+1
        DO j=i,n
         k=k+1
         veca(k)=matrixa(k2,j)
         IF(i==j)veca=REAL(i)
       END DO
      END DO
      vecz=0.0d0
! now check memory is sufficient
      index=1
      ALLOCATE(iscratch(3*n),STAT=ier)
      CALL fmemreq(index,n,mapA,mapZ,mapZ,isize,rsize,ptrsize, &
     &     iscratch)
      DEALLOCATE(iscratch)
      WRITE(6,'('' isize'',i6)')isize
      WRITE(6,'('' rsize'',i8)')rsize
      WRITE(6,'('' ptrsize'',i6)')ptrsize
      ALLOCATE(iscratch(isize),STAT=ier)
      ALLOCATE(dptr(ptrsize),STAT=ier)
      ALLOCATE(scratch(rsize),STAT=ier)
      IF(ier/=0)THEN
        WRITE(6,'('' ***** ier ***** '',i4)')ier
        CALL abort()
      END IF
      ivector=0
      irange=1
      lb=1.0d0
      ub=0.0d0
      ilb=-1
      iub=-1
      abstol=0.0d0
      start=RTC()
! to select eigenvalues/vectors
!      CALL pdspevx(ivector,irange,n,vecA,mapA,
!     &     lb,ub,ilb,iub,abstol,meigval,vecZ,mapZ,eval,
!     &     iscratch,isize,dptr,ptrsize,scratch,rsize,info)
! to get all eigenvectors and values
      CALL pdspev(n,veca,mapa,vecz,mapz,eval, &
     &     iscratch,isize,dptr,ptrsize,scratch,rsize,info)
      IF(info/=0)THEN
        WRITE(6,'('' ***** error in pdspev *****'')')
        WRITE(6,'('' info = '',i4)')info
      ELSE
        WRITE(6,'('' *****  pdspev successful *****'')')
      END IF
```

```
      finish=RTC()
      time=finish-start
      WRITE(6,'('' time to do n = '',i6,'' is '',g12.5,'' seconds'')')n,time
! copy results from vecz into matrixa if required
      k=0; k2=0
! for each row
      DO i=1,n
        IF(mapz(i)/=inode)CYCLE
        k2=k2+1
        DO j=i,n
          k=k+1
          matrixa(k2,j)=vecz(k)
        END DO
      END DO
! now concatenate matrixa on every node
!       ...
      END
```

2.3.2 Benchmark results

Table 5: Performance of PeIGS2.1 Routine PDSPEV on the Cray T3D

Order of	Number of Nodes [all times in seconds]						
matrix	4	8	16	32	64	128	256
64	0.3398	0.4606	0.5208		1.444		
128		0.8613	0.7955		2.050		
256		2.666	2.176		3.532		
512		11.44	8.347	8.236	8.038		
1024		75.92	52.67	37.52	32.18		
2048			383.1	253.4	195.9	173.9	
4096					2195		

It seems from this table that for reasonably large problems PeIGS is significantly slower than the similar ScaLAPACK routine on the T3D. This is because the latter uses the Cray implementation of the BLACS which use the SHMEM primitives which have extremenly low latency, whereas PeIGS is using MPI (see appendix). This is particularly obvious in the much improved scalability of the ScaLAPACK routine. Moreover PeIGS has to do a modified Gram-Schmidt procedure which involves extra communication. This indeed makes its performance a little slower than EISCUBE, but not much.

Figures in parentheses probably indicate the system was paging. We note that indeed PeIGS is beating ScaLAPACK on the SP2 where the communication has higher latency and block-cyclic data decomposition is not an advantage.

Table 6: Performance of PeIGS2.1 Routine PDSPEV on IBM SP2

Order of matrix	Number of Nodes [all times in seconds] 1	2	4	8	16
512	17.5	10.5	8.07	7.26	
1024	107.0	54.6	34.0	28.27	
2048		(786.9)	286.1	200.3	
4096				(3882)	

2.4 HJS Algorithm

Hendrickson, Jessup and Smith of Sandia National Laboratory, USA developed another eigensolver for dense symmetric systems [35]. This was done specifically for the Intel Paragon, but the code could be ported to other machines easily.

It is closely related to the method used in ScaLAPACK, but differs in the blocked cyclic distribution pattern. Instead of arbitrary rectangular blocks they chose to use the simpler case of square blocks. They also distributed the blocks in an element-wise square torus mapping rather than a block-wise torus mapping as in ScaLAPACK.

A significant enhancement however is flexibility of algorithmic blocking. In ScaLAPACK the size of computational blocks is the same as the data blocks d$\times$d. In HJS however the data blocks are only one element each, but the number of elements used in computation and communication is specified separately. This premits optimisation by allowing cache reuse to be controlled.

Orthogonality of eigenvectors is ensured in this routine using a modified Gram-Schmidt procedure on clustered values [43].

We have not yet obtained the code, but its performance is compared to ScaLAPACK on the Sandia's Intel Paragon in the original paper [35]. It appears that in the tridiagonalisation step, for optimal block size, the HJS algorithm is some 30% faster that ScaLAPACK for large matrices, but roughly the same in the back-transformation step.

3 Jacobi-type iterative methods

The method works by repeated application of orthogonal Jacobi rotations, which, depending on the order of them, bring the matrix more or less quickly to diagonal form. The original method used by Jacobi in 1846 applied the rotations to zero the largest non-zero off-diagonal element, but on a computer a re-designed pattern is easier. The eigenvector matrix is just the product of all the Jacobi matrices. This is true when a two-sided similarity transform is used but a one-sided variant is known to be more efficient in a parallel implementation. A one-sided tridiagonalisation, rather than the usual Householder reflection, was described by Hegland [33].

It is also well known that Jacobi eigensolvers improve as the matrix is more diagonally dominant, quite simply there are less iterations to achieve the required accuracy. This makes them particularly suitable to applications which are of a 'perturbative' nature; that is, a solution is already known to a closely similar problem. This is the case in certain classes of optimisation problem where a solution is followed in time or in configuration space.

On serial machines the two-sided algorithm is simply:

$$\begin{aligned} \mathbf{A}_{k+1} &= \mathbf{J}_k^T \mathbf{A}_k \mathbf{J}_k \\ &= \mathbf{V}_{k+1}^T \mathbf{A} \mathbf{V}_{k+1} \end{aligned}$$

Direct implementation of this on a parallel machine can cause problems. Pre-multiplication by the rotation matrix updates the *rows* of $\mathbf{A}$, but post-multiplication by the rotation matrix updates the *columns* of $\mathbf{A}$. So there is no natural data distribution which fits both the multiplications. This can result in a lot of communications. A one-sided method is likely to be better for this reason.

In the sequential implementation rotations are typically skipped if the value to which they would be applied is less than a threshold. This is difficult to schedule in parallel.

3.1 A one-sided Jacobi code – the BFG routine

Ian Bush implemented a one-sided Jacobi algorithm as now described. Following Littlefield and Maschhoff [48] we can factor $\mathbf{A}_k$ as

$$\mathbf{A}_k = \mathbf{G}_k \mathbf{V}_k$$

where

$$\mathbf{G}_k = \mathbf{V}_k^T \mathbf{A}$$

Then at each step, given a rotation matrix $\mathbf{J}_k$, we can update the two factors by

$$\begin{aligned} \mathbf{G}_{k+1} &= \mathbf{J}_k^T \mathbf{G}_k \\ \mathbf{V}_{k+1} &= \mathbf{V}_k \mathbf{J}_k \end{aligned}$$

This does not involve the explicit formation of $\mathbf{V^T A}$ from $\mathbf{A}$ at each step but only a rotation of rows of $\mathbf{G}$. Remaining off-diagonal elements of $\mathbf{A}_k + 1$ must be eliminated as we go along.

To calculate $\mathbf{J}$ to eliminate say A_{pq} we calculate

$$A_{pq} = \mathbf{G}_{p\bullet} \cdot \mathbf{V}_{\bullet q}$$

and then use the serial technique to get the rotation angle.

3.1.1 What do we gain by this ?

The updates for eliminating A_{pq}

$$\begin{aligned}\mathbf{G}_{k+1} &= \mathbf{J}_k^T \mathbf{G}_k \\ \mathbf{V}_{k+1} &= \mathbf{V}_k \mathbf{J}_k\end{aligned}$$

act *only* on *rows* p and q of $\mathbf{G}_k$ and *columns* p and q of $\mathbf{V}_k$.

Threrefore steps taken are

1. Distribute $\mathbf{G}$ by rows and $\mathbf{V}$ by columns – all local eliminations require no communications
2. Each processor can then work simultaneously on eliminating different elements as the rotations don't interfere with each other

The only other required component for the implementation is an efficient shuffling scheme for $\mathbf{G}$ and $\mathbf{V}$ between the processors such that each element of $\mathbf{A}$ is eliminated once per sweep.

3.1.2 Pros and cons

The one-sided Jacobi method requires less communications than the traditional two-sided Jacobi or Householder reduction methods so hopefully it will scale better. It however requires of the order n^2 extra dot-products over and above the requirements for the two-sided Jacobi.

Furthermore serial Jacobi diagonalisers are normally around three times slower than those using Householder techniques. Thus the serial speed of a one-sided Jacobi is known to be poor unless the matrix is nearly diagonal at the start.

The performance of this diagonaliser is thus a trade off between better scaling and poor single node speed. To be practical it must scale much better than the other methods. This is further discussed below

3.1.3 The algorithm in its full glory

Initialize $\mathbf{G}_{p\bullet}$ and $\mathbf{V}_{\bullet p}$ for all p assigned to this processor
Split the set of p's owned by this processor into Group I and Group II
Loop to convergence
    – First do the Eliminations
    For all pairs of p and q selected from Group I eliminate A_{pq}
    For all pairs of p and q selected from Group II eliminate A_{pq}
    For round = 1, ... $2\times p$ - 1

```
            If Group I > Group II
                  Loop over all p's in Group I
                        Loop over all q's in Group II
                              Eliminate A_pq
                        End Loop over Group II
                  End Loop over Group I
            Else If Group II >= Group I
                  Perform above loops switching Groups I and II
            Endif
            – Now do the Shuffling
            If round is even and processor number = round / 2
                  Send/receive Group II i.e. Send Group II to the
                  next processor and recieve a new Group II from the
                  previous one
            Else
                  Send/receive Group I
            Endif
      End For round
      Check Convergence
End Loop to Convergence
```

The elimination step

This is fairly standard apart from the dot products

– Reconstruct Required Elements of **A**
$A_{pq} = \mathbf{G}_{p\bullet} \cdot \mathbf{V}_{\bullet q}$
$A_{pp} = \mathbf{G}_{p\bullet} \cdot \mathbf{V}_{\bullet p}$
$A_{qq} = \mathbf{G}_{q\bullet} \cdot \mathbf{V}_{\bullet q}$
– Compute Rotation Angle
$\theta = (a_{qq} - a_{pp})/(2a_{pq})$
$t = sgn(\theta)/(\mid \theta \mid + \sqrt{(\theta^2 + 1)})$
$c = 1/\sqrt{(t^2 + 1)}$
$s = tc$
– Update G and V
For i = 1, ... order of Matrix
  $T = G_{pi}$
  $G_{pi} = cT - sG_{qi}$
  $G_{qi} = sT + cG_{qi}$
  $T = V_{ip}$
  $V_{ip} = cT - sV_{iq}$
  $V_{iq} = sT + cV_{iq}$
End For i

Optimisations

Several steps are possible:

1. Rather than store $\mathbf{G}$, store $\mathbf{G}^T$ – unit strides are then possible in the dot products and update loop;
2. Combining the two elimination steps outside the `round` loop with the `round = 1` iteration gives a standard Jacobi on the local data. Since this corresponds to a square block on the diagonal of the matrix the symmetry of the matrix can then be exploited;
3. The dot products to generate the diagonal elements of $\mathbf{A}$ can be done outside the elimination loops and these elements can be updated as the process proceeds. This saves two dot products in the elimination step;
4. Suppose the outer loop in the elimination stage is over p. Then once all the updates on $\mathbf{G}_{p\bullet}$ and $\mathbf{V}_{\bullet p}$ have been done they can be sent onto the next processor. Even better this can be done asynchronously while step p + 1 is being performed. On a low latency machine (e.g. the T3D) this was found to improve scalability.

All these improvements are done in the current version of the code.

The data is distributed by blocks of columns. A mapping array allows different processors to hold different numbers of columns, but the routine assumes that the block allocated to a processor is at least two elements wide. The solver will also optionally rank the eigenvalues, and the user may provide an initial guess at the eigenvectors.

3.1.4 Benchmark results

The algorithm as it stands requires that at least two columns of the matrix are stored on each processor. This may cause problems with small matrices on large MPP machines. For instance on a 256 processor machine the smallest matrix you can diagonalise by this method is order 512.

It was however found in the original paper [48] that the BFJ method is the most suitable for solving problems where the number of processors p is comparable with the order of matrix n.

The way to implement it is also described in the paper. A secondary layer of distribution is required such that the columns of the matrices are themselves distributed over a number of processors. About the only extra complication is that the dot products have to be distributed. This extra stage has not, as yet, been implemented.

In an analysis of the computational complexity of the BFJ method Littlefield and Maschhoff came to the important conclusion that message-passing startup time (latency) would be the most important factor to limit calculations. We thus see an enormous advantage in using the Cray T3D as opposed to a machine like the IBM SP2 for this reason.

3.2 A two-sided Jacobi code – the Pittsburgh PJAC routine

Another Jacobi algorithm is available from the University of Pittsburgh. It is currently able to handle complex symmetric or Hermitian matrices as well as real symmetric ones. The PJAC library was

Table 7: Performance of Routine BFG on the Cray T3D

Order of	Number of Nodes [all times in seconds]						
Matrix	4	8	16	32	64	128	256
64	0.0875	0.0593	0.0454	0.0504			
128	0.6534	0.370	0.2184	0.1607			
256	5.693	2.95	1.554	0.8766			
512	61.277	31.5	15.88	7.981	4.689		
1024	756.5	368.7	186.23	95.58	56.319	29.62	17.222
2048					499.76	264.31	144.18

Table 8: Performance of Routine BFG on IBM SP2

Order of	Number of Nodes [all times in seconds]				
Matrix	1	2	4	8	16
64	0.09	0.05	0.04	0.05	0.06
128	0.65	0.37	0.24	0.2	0.17
256	5.11	2.88	1.69	1.03	0.83
512	44.97	24.48	13.62	7.98	5.11
1024	408.54	222.32	132.35	64.31	39.15

written by D.C.O'Neal and R.Reddy, PSC Parallel Applications Group. E-mail *oneal@psc.edu* and *rreddy@psc.edu* . This diagonaliser routine was developed from a project code written for the 1994 Institute on Parallel Computing. The algorithm employed is essentially a hybrid of Jacobi methods. Problem sizes ranging from n = 2 to 8192 have been solved.

The PJAC library provides a scalable eigensolver for real/complex, symmetric/Hermitian matrices. File handling routines for creating input systems and viewing output systems are bundled with the solver.

The algorithm employed is again a block-factored Jacobi, but this time the traditional two-sided variety. The data distribution is by square blocks, each of which must be the same size. This requires the order of the matrix to be an exact multiple of the number of nodes used to diagonalise it, which could be inconvenient.

The parallel Jacobi method described by Brent and Luk assumes the existence of a two-dimensional grid of $(n/2)^2$ processors to resolve an n×n matrix; i.e. the input system is mapped onto a collection of 2×2 elements. A variation of this method featuring virtualised elements was implemented first.

Performance and scaling of the virtualised Brent-Luk method did not work well for larger problems on the Cray T3D. Essentially, the grain size associated with the algorithm was found to be too small. Time spent fetching data points could not be offset by cache reuse.

This led the authors to work on a generalisation of the Brent-Luk work in which the grain size was made a function of n and the processor count. As the processor count ranges from 1 to n/2, their macro-element Jacobi method varies from the serial Jacobi method to the Brent-Luk parallel model. Good performance and scaling was then observed for a fairly wide range of element dimensions.

Documentation for PJAC can be found on the Web at URL *http:///general/software/cray/t3d/pjac/pjac.html* and *http://www.psc.edu/htbin/man_search?LOCAL/rspjac* .

A paper describing this work was presented at the Autumn 1994 Cray User Group Conference in Tours, France. Copies may be obtained by writing to the authors.

3.2.1 Benchmark results

Ian Bush took steps to improve the performance by adding a convergence check. This reduced the execution times to around 60% of the original for the test cases. We note the original routine publicised by PSC still does not contain this modification.

Table 9: Performance of Improved pJac on the Cray T3D

Order of	Number of Nodes [all times in seconds]						
matrix	4	8	16	32	64	128	256
16	0.021						
32	0.041						
64	0.108						
128	0.555						
256	6.026						
512	52.455	20.818	8.860	5.835	7.972		
1024	591.17	194.64	77.292	38.106	29.736	37.75	60.551
2048				301.638	169.52	134.69	179.2
4096					1290.0	565.79	
8192					10857.5	4951.4	3542.0

3.3 Parallel NAG library

The parallel NAG library, a subset of the full NAG library [51], is initially targeted at distributed-memory machines with the PVM message-passing harness. It will be converted to use MPI sometime in the future, and has been designed with HPF in mind. The logical topology assumed is a 2D grid so PBLAS, BLACS and ScaLAPACK can be used as building blocks. Distribution of data and collection of results is the responsibility of the user, but some routines are provided to help. Both a Users' Manual and on-line documentation exist and NAG have a WWW page:

http://www.nag.co.uk .

The contents of the parallel NAG library include:

- some ScaLAPACK routines
- drivers based on ScaLAPACK
- symmetric eigenproblem and single-value decomposition (SVD)
- sparse solvers

- 1D, 2D and multi-D quadrature
- unconstrained nonlinear least squares solver
- random number generator
- data distribution support routines
- utilities (error handling etc.)

The symmetric eigenproblem solver is based on a one-sided Jacobi method in routine F02FQFP.

4 Conjugate-gradient type iterative methods

Other methods are useful in the case of very large sparse systems. In this section we consider Lanczos, Arnoldi and Subspace Decomposition Methods which are mainly used for solving systems of linear equations but are also applied to the eigenvalue prroblem.

Lanczos methods are described by Cullum and Willoughby [16]. The original reference is [45]. You might be interested in reading an evaluation of sequential software for computing eigenvalues of sparse nonsymmetric matrices [47]. We only know of one parallel package, PARPACK.

4.1 PARPACK

A parallel version of the ARPACK library is now available [46]. The message-passing layers currently supported are BLACS and MPI. PARPACK is provided as an extension to the current ARPACK library (Release 2.1).

More detailed information about Parallel ARPACK is available in an extended abstract [52] which can be obtained from *http://www.netlib.org/scalapack/parpack/cm_write.html* .

4.1.1 ARPACK software

ARPACK is a collection of Fortran 77 subroutines designed to solve large scale eigenvalue problems.

The package is designed to compute a few eigenvalues and corresponding eigenvectors of a general n by n matrix A. It is most appropriate for large sparse or structured matrices A where structured means that a matrix-vector product w $\leftarrow$ Av requires order n rather than the usual order n^2 floating point operations. This software is based upon an algorithmic variant of the Arnoldi process called the Implicitly Restarted Arnoldi Method (IRAM). When the matrix A is symmetric it reduces to a variant of the Lanczos process called the Implicitly Restarted Lanczos Method (IRLM). These variants may be viewed as a synthesis of the Arnoldi/Lanczos process with the Implicitly Shifted QR technique that is suitable for large scale problems. For many standard problems, a matrix factorisation is not required. Only the action of the matrix on a vector is needed.

ARPACK software is capable of solving large scale symmetric, nonsymmetric, and generalised eigenproblems from significant application areas. The software is designed to compute a few (k) eigenvalues with user specified features such as those of largest real part or largest magnitude. Storage requirements are of the order of n×k locations. No auxiliary storage is required. A set of Schur basis vectors for the desired k-dimensional eigen-space is computed which is numerically orthogonal to working precision. Numerically accurate eigenvectors are available on request.

Important features are:

- Reverse communication interface;
- Single and double precision real arithmetic versions for symmetric, non-symmetric;
- Standard or generalized problems;
- Single and double precision complex arithmetic versions for standard or generalised problems;
- Routines for banded matrices – standard or generalized problems;
- Routines for the singular value decomposition;
- Example driver routines that may be used as templates to implement numerous shift-invert;
- Strategies for all problem types, data types and precision.

The performance of these routines is clearly problem specific because of the need for a matching matrix-vector product code. We have therefore not performed any benchmark tests.

4.2 SYISDA – Symmetric Invariant Subspace Decomposition Algorithm

SYISDA version 2.0 implements the symmetric invariant subspace decomposition algorithm which was developed by members of the PRISM (Parallel Research on Invariant Subspace Methods) project: Christian Bischof, William George, Steven Huss-Lederman Xiaobai Sun, Anna Tsao, G.Zhang and Thomas Turnbull of Argonne National Laboratory, the Center for Computing Sciences, Bowie Maryland and Duke University USA [39, 40].

SYISDA is a divide-and-conquer algorithm which derives most of its parallelism from performing matrix multiplication [5]. The basic algorithm was presented by Lederman et al. [39]. It can be briefly stated as follows:

1. Suppose A is a dense n×n matrix and Z is to be the eigenvector matrix (if desired) initialised to zero.
2. Compute upper and lower bounds on the spectrum of A and for l(A) where l is a linear function that maps spectrum A into [0,1] so that its mean eigenvalue is mapped to 0.5.

3. Iterate:
   $A_0 = l(A)$
   $A_k = P_k(A_{k-1}), k = 1, 2, 3...$
   where the polynomials P_k are designed so that the eigenvalues of the sequence of iterates tend toward wither 0 or 1. Iterations terminate when the sequence of matrices converges.

4. Compute the two independent subproblems of A, $A_1 = Q_1^t A Q_1$ and $A_2 = Q_2^t A Q_2$, where $Q = [Q_1 Q_2]$ is an orthogonal matrix such that the columns of Q_1 and Q_2 are a basis for the range and null space of the iterated matrix respectively. If eigenvectors are desired Z must be updated to ZQ.

Since A_1 and A_2 are independent the algorithm can be applied recursively until all subproblems are diagonal or may be treated sequentially on one processor.

Step 2 in the algorithm consists entirely of DGEMM matrix primitives and dominates the computational cost. Whilst SYISDA is significantly more expensive in computational complexity than an equivalent QR algorithm, multiplication is implemented efficiently on most current parallel hardware and sequentially yields highest obtainable performance on widely-used processor architectures. Resulting preformance is however 3-6 times slower than symmetric QR in current implementations.

The SYISDA package is built on LAPACK and BLAS and uses MPI for its point-to-point and global communications. It contains, as its kernel routine, a parallel matrix multiply package with similar functionality to DGEMM which uses the broadcast-multiply-roll algorithm [30]. It works on any matrix size on an arbitrary number of processors treated as a square 2D topology. Most of the time taken by the SYISD algorithm is indeed in sub-matrix multiplication.

Storage of data in the input matrix A and eigenvector matrix Z is in the format of a wrapped torus around the square 2D processor grid. This is equivalent to HPF data storage.

A full set of published technical memoranda and working notes as well as an installation guide is available from the anonymous ftp server *ftp://ftp.super.org/pub/prism* .

4.2.1 Benchmark results

Given the relatively low preformance of the SYISDA package at the time of writing we only show limited SP2 results. A port to the T3D would need to be done if it were required on that machine.

Table 10: Performance of PRISM routine SYISDA on IBM SP2

Order of	Number of Nodes [all times in seconds]			
matrix	1	4	9	16
128		10.3 (ip)		
256		33.1 (ip)		
512		155.6 (ip)		
1024		359.4		

(ip) – indicates job run on shared `ip` network to test

4.3 Another parallel divide and conquer algorithm – Potter's code

Cuppen's [17] divide and conquer algorithm allows the solution of eigenproblems in a faster way than the usual QR and, more importantly, is easily suitable for parallelisation. There are problems of orthogonality in pathological cases but solutions to these have been found [56].

An early version of this code was obtained from Chris Potter of the EPFL Lausanne PATP group. The algorithm employed [56] is reduction to tridiagonal form (using a parallel Householder code) and then the divide and conquer method of Dongarra and Sorensen [23]. Unfortunately the code as it stands has a number of deficiencies which make its use in an actual application difficult. First, as with the PJAC routine, the order of the matrix must be a multiple of the number of processors. This is so that it can be divided into sub-matrices of size $n/p \times n/p$ to simplify the implementation. More serious is the restriction to matrices $n \leq 2048$ at present, as it was only envisaged to treat small matrices at EPFL. This is because of memory usage to hold replicated data. Furthermore, only the eigenvectors of the tridiagonal matrix are returned so a back-transformation step is required, but not provided. Since only the object code is available it can not be modified to work around these problems, in particular the restriction to small matrices.

Anyone interested in using this method, or in a modified version for larger systems, should contact Chris Potter directly *potter@patpserv.epfl.ch* .

4.3.1 Benchmark results

These results were taken from Potter's article [56], but as noted above do not include time for the back transformation.

Table 11: Performance of Potter's code on T3D

Order of	Number of Nodes [all times in seconds]						
matrix	2	4	8	16	32	64	128
512	24.7	11.73	6.06	3.29	1.93	1.22	0.92
1024	174.8	82.0	41.2	21.4	11.7	6.8	4.44
2048			301	153.5	80.3	43.5	25.2

4.4 Spectral Transform Lanczos Method

We await a copy of this code to test. It was developed by Soren Paedkjar of the California Institute of Technology [54].

5 Acknowledgements

RJA thanks Dr.M.Terao-Dunseath and the Université de Rennes I for a position as Visiting Professor during 1 month in 1995-96.

We thank Drs. Ken Stanley and Patrick Briddon for useful discussions on the ScaLAPACK and BLACS routines.

6 Computer systems

The following benchmark platforms were available for test purposes

6.1 IBM SP2

Timings were taken on a variety of systems

- 512-node machine at the Cornell Theory Center
- 79-node machine at CNUSC, Montpellier
- 16-node machine at Daresbury Laboratory (DL)
- 4-node machine at Laboratoire SIMPA, Université de Rennes I

Nodes were 66.7 MHz RS/6000 RIOS-2 processors, of the "thin" variety at Cornell and the TN2-nodes at Daresbury and in France. The French systems were running AIXv4.1.

The IBM SP2 at Daresbury consists of 14 thin node 2 (TNT) processors, each having 64 Mbytes of memory and 2 wide nodes, each with 128 Mbytes of memory. All nodes have 128 Kbytes of level 1 data cache and 32 Kbytes of instruction cache. The processors standard Power2 Architecture RS/6000 processors capable of four floating-point arithmetic instructions per clock cycle providing the quad-load pipes can keep the cache filled. The SP2 nodes therefore yield a peak performance of 267 Mflops. SP2 nodes at CNUSC and SIMPA have 256MB memory.

The interconnect between the nodes is a high performance switching network capable of sustaining a high bandwidth. In addition each node has an Ethernet connection and the two wide nodes an FDDI connection. All nodes have fast wide SCSI II discs attached for scratch space and access to permanent disc store is via NFS, using the wide nodes and FDDI connections. The wide nodes are used for login, editing and compilation as well as computation.

The high-performance switch configuration at Cornell is worthy of note, as it is somewhat nonstandard due to the unusually large configuration installed there. The additional switching hardware required may be expected to degrade message-passing performance when compared to a smaller configuration.

Compilation options

```
xlf -O3 -qhot -qnosave -qarch=pwr2 -lpesslp2 -lblacsp2 -lesslp2
```

6.2 Cray T3D

Timings are from the 512-processor T3D at EPCC. The Cray T3D system uses DEC alpha EV4 nodes with 151 MHz clock therefore rated at 151 MFlops peak floating points arithmetic performance. The use of a direct-mapped cache with only a single read-ahead buffer makes some code re-structuring essential for optimisation.

Compilation options
`cf77 -c -Ccray-t3d -O1 -Oscalar3` with BLAS library and level-2 and a level-3 Fortran code
`cf77 -c -Ccray-t3d -Wf''-o unroll''` with Fortran BLAS level-1

6.3 iPSC/860

Timings are from the 64-processor Intel iPSC/860 at Daresbury.

Compilation options
`if77 -O4 -Mvect -lclasspack`

6.4 Parsytec GC

The Parsytec GC at The University of Paderborn, Germany consists of 64 nodes, each comprising two PowerPC 601 processors and T9000 transputer based switching hardware. Message-passing was performed using Parsytec's PowerPVM implementation, GA-Tools was ported to the Parsytec by dedicating one of the two processors per node to global memory management, with communication using PowerPVM. 64Mb of memory are available on each node.

6.5 SGI PowerChallenge Array

Timings were collected from the European Power Challenge Array at Cortailloid in Switzerland. All nodes used for benchmarking were R8000 processors running at 75 MHz with a main-memory configuration of 2-2.5 Gb. Message passing is performed using TCGMSG and GA-Tools, which within a node is implemented using shared-memory, and between nodes uses TCP-IP to communicate over the HIPPI switch (the latter capability has not yet been benchmarked).

6.6 HP Workstation cluster

The HP workstation cluster at Daresbury consists of one HP/9000-755 workstation with 128 Mb memory, and 4 HP/9000-735s with 80 Mb memory. They are connected by ethernet and FDDI networks, the latter being used for the tests reported here.

6.7 IBM Workstation cluster

The IBM workstation cluster at Daresbury consists of six IBM RS/6000. Tci11-tci13 are three model 530H systems (approx. 50M Flops) and tci14-tci16 are three model 370 systems (approx. 100 MFlops). They are connected by ethernet, FDDI and ATM networks. Tci12 is not accessible for running user code but acts as a file server.

Compilation options
`xlf -O3 -qhot -qnosave -qarch=pwr2`

6.8 Cray J932-4096

The Rutherford Appleton Laboratory 32-processor Cray J90 was used. The Cray J932-4096 is a shared memory, vector computer manufactured by Cray Research Inc. Each processor has a clock cycle of 10 nsec and chained add and multiply vector pipes giving 200 Mflop per node and an overall performance of 6.4 Gflops with a memory bandwidth capable of sustaining peak performance. The 4 Gbyte main memory is a "flat" high speed memory for maximum vector performance, unlike earlier Cray computers the J90 has a small cache memory for instructions and scalar data only.

Compilation options
`cf77 -Oscalar3 -Ovector3 -Otask3`

6.9 PowerPC model 250T

Tci17 is a 66 MHz PowerPC RISC/6000 model 250T (approx. 50 MFlops).

Compilation options
`xlf -O3 -qhot -qnosave -qarch=ppc`

7 Message-passing performance

The performance of the message-passing system can be characterised using the parameters of Hockney and Carmona [36], namely

- r_∞, the asymptotic transfer rate in MB/s and
- $n_{\frac{1}{2}}$, the message length in Bytes at which half this rate is achieved.

A variety of message passing harnesses have been investigated, including MPL, MPI, PVM3, PVMe, TCGMSG, GA-Tools, HPF and BSP. MPICH (Argonne National Laboratory, USA) has also been tested but since a bug was discovered on the T3D no results can be reported. The CRI/EPCC implementation of MPI is described in a paper by Cameron et al [11] andn on the WWW at URL *http://www.epcc.ed.ac.uk/t3dmpi/Product/* . A full description of these tests are presented in a separate report [3]. At this point we merely summarise these results through the tables below, which collate the Hockney and Carmona parameters for the different software and networks investigated.

These parameters will be used throughout this document in our discussion of the performance of each application and, together with the sequential node performance of the machine, are critical in an overall characterisation and parallel scaling model.

Table 12: The IBM SP2 †

software	protocol	r_∞	$n_{\frac{1}{2}}$	latency [microsecs]
MPL	ip	8.38	3037	270
MPL	us	35.2	3032	86
MPICH	ip	6.26	4727	656
MPICH	us	34.8	3966	114
MPICHv4	ip	10.28	3853	599
MPICHv4	us	33.05	2939	142.7
MPIv4	ip	10.24	3829	605.6
MPIv4	us	32.8	2965	153.0
PVMe	us	34.14	18297	535
TCGMSG v4.01	us	34.8	3062	88

† We do not differentiate between wide and thin nodes as they are essentially the same.

Table 13: The Cray T3D

software	protocol	r_∞	$n_{\frac{1}{2}}$	latency [microsecs]
shmem_put	shmem	120.2	780	6
shmem_get	shmem	58	362	6
PVM	shmem	25.7	3357	138
TCGMSG	shmem	47.5	1729	36.4
MPI (CRI/EPCC)	shmem	50.4	2662	50

The GA-Tools developed by Robert Harrison and Rick Littlefield at Battelle Pacific Northwest National Laboratory, USA [53], are available on both the T3D, SP2 and other systems at Daresbury. They provide an easy interface to virtual shared memory for the storage and asynchronous access of large matrices. Some global operations are provided too. At this point we refer the reader to a rather thorough report by Howard Pritchard [58] that benchmarks most functionality of the GA-Tools on a Cray T3D, Intel Paragon and IBM SP1 and SP2.

Other software and detailed performance tests are described in [3]. Many of these tests were carried out using an 'unsupported' version of the MPI (Message Passing Interface) environment. Its point-to-point bandwidth is satisfactory, as reported in [3], since it is built on top of MPL. However we await the supported IBM release of the standard MPI software with the release of Parallel Environment V2.1.

In practise, a number of machines are still lacking efficient MPI routines, including the SP2. This is particularly important for some of the applications requiring global reduction routines, e.g. implicit solvers. Only naive implementations of global operations are available in most cases in order to achieve the functionality of the standard as early as possible. To overcome this problem, some applications are still using hand-written communication routines, for this purpose. The full (and very rich) MPI functionality for global operations will be used when it is available in an efficient form.

References

[1] *http://www.ucar.edu/SOFTLIB/mathlib.html*

[2] Available from *ftp://ftp.super.org/pub/prism/prism_syisda.tar.Z*

[3] R.J.Allan and M.F.Guest *Parallel Application Software on High Performance Computers: The IBM SP2 and Cray T3D* (Daresbury Laboratory, 1996)

[4] E.Anderson, Z.Bai, C.Bishof, J.Demmel, J.Dongarra, J.Du Croz, A.Greenbaum, S.Hammarling, A.McKenney, S.Ostrouchov and D.Sorensen *LAPACK Users' Guide* 2nd edition SIAM (1995)

[5] L.Auslander and A.Tsao *On parallelizable Eigensolvers* Adv. Appl. Math. 13 (1992) 253-61

[6] R.Barrett, M.Berry, T.Chan, J.Demmel, J.Donato, J.Dongarra, V.Eijkhout, R.Pozo, C.Romine and H. van der Vorst *Templates for the solution of linear systems: building blocks for iterative methods* (SIAM, 1993) also available from *anonymous@ftp://netlib.ornl.gov*

[7] R.H.Barlow and D.J.Evans *A parallel organisation of the bisection algorithm* The Computer Journal 22 (1977) 267-69

[8] R.H.Barlow, D.J.Evans and J.Shanehchi *Parallel multisection applied to the eigenvalue problem* The COmputer Journal 26 (1983) 6-9

[9] H.J.Bernstein and M.Goldstein *Parallel implementation of bisection for the calculation of eigenvalues of tridiagonal symmetric matrices* Computing 37 (1986) 85-91

[10] H.J.Bernstein and M.Goldstein *Optimising Givens' algorithm for multiprocessors* SIAM J. Sci. Stat. Computing 9 (1988) 601-2

[11] K.Cameron, L.J.Clarke and A.G.Smith *CRI/EPCC MPI for the Cray T3D* (EPCC, 1995)

[12] H.Y.Chang, S.Utku, M.Salama and D.Rapp *A parallel Householder tridiagonalisation stratagen using scattered square decomposition* Parallel Computing 6 (1988) 297-311

[13] H.Y.CHang, S.Utku, M.Salama and D.Rapp *A parallel Householder tridiagonalisation stratagen using scattered row decomposition* Intl. J. Num. Meth. Engine. 26 (1988) 857-73

[14] J.Choi, J.Dongarra, R.Pozo and D.Walker, *ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers* in Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, IEEE Comput. Soc. Press (1992)

[15] J.Choi, J.Dongarra and D.Walker *The design of a parallel dense linear algebra software library: Reduction to Hessenberg, tridiagonal and bidiagonal forms* Numerical Algorithms 10 (1995) 379-400

[16] J.Cullum and R.A.Willoughby *Lanczos and the computation of eigenvalues in specified intervals of the spectrum of large, sparse, real symmetric matrices* in Sparse Matrix Proceedings 1978 (SIAM, 1978)

[17] J.J.M.Cuppen *A divide and conquer method for the symmetric tridiagonal eigenproblem.* Numer. Math. 36 (1981) 177-95

[18] G.Dahlquist et al. *Numerical methods* (Prentice Hall 1974)

[19] J.Demmel and K.Stanley *The Performance of finding eigenvalues and eigenvectors of dense symmetric matrices on distributed-memory computers* preprint, University of California, Berkeley (1994)

[20] J.W.Demmel, M.T.Heath, H.A.van der Vorst *Parallel Numerical Linear Algebra* LAPACK Working Note 60, Computational Science Reports CS-93-192 University of Tennessee, Knoxville (1993)

[21] J.Dongarra, J.Du Croz, S.Hammarling and Hanson ACM Trans. Math. Software 14 (1988) 1-32

[22] J.Dongarra, J.Du Croz, I.S.Duff and S.Hammarling ACM Trans. Math. Software 16 (1990) 1-28

[23] J.Dongarra and D.Sorensen *A fully parallel algorithm for the symmetric eigenproblem* SIAM J. Sci. Comput. 8 (1987) 139-54

[24] J.Dongarra and R.V.de Geijn *LAPACK working note number 30: Reduction to condensed form for the eigenvalue problem on distributed memory architectures* (Dept. of Computer Science, University of Tennessee, 1991)

[25] J.Dongarra et al. *LINPACK Users' Guide* SIAM (1979)

[26] I.S.Duff et al. *Catalogue of subroutines* Harwell report (June 1993)
I.S.Duff et al. *Releae 11 specification* Harwell report Vol 1. (June 1993) Vol 2. (July 1993)

[27] D.Elwood, G.Fann and R.Littlefield *Parallel Eigensolver System* User Manual available from *anonymous@ftp://pnl.gov*

[28] G.Fann, D.Elwood and R.Littlefield *Performance of a fully parallel dense real symmetric eigensolver in quantum chemistry applications* PNL preprint

[29] G.Fann and R.J.Littlefield *Parallel inverse iteration with reorthogonalisation* in Proc. 6th SIAM conference on Parallel Processing for Scientific Computing (SIAM, 1993) pp409-13

[30] G.Fox, S.Otto and A.J.G.Hey *Matrix algorithms on a hypercube I: matrix multiplication* Parallel Computing 4 (1987) 17-31

[31] T.L.Freeman and C.Phillips *Parallel Numerical Algorithms* Prentics Hall (1992) ISBN 0-13-651597-5

[32] G.H.Golub and C.F.van Loan *Matrix Computations* (John Hopkins University Press, 1983)

[33] M.Hegland *A distribution independent algorithm for the reduction to tridiagonal form using one-sided rotations* Proc. 1st IEEE Int. Conf. on Algorithms and Architectures for Parallel Processing (IEEE, Computer Society Press, 1995) 286-9

[34] B.A.Hendrickson and D.Womble *The torus-wrap mapping for dense matrix calculations on massively parallel computers* SIAM J. Sci. Comput. 15 (1994) 1201-26

[35] B.A.Hendrickson, E.Jessup and C.Smith *A Parallel Eigensolver for Dense Symmetric Matrices* Available from *ftp://ftp.cs.sandia.gov/pub/papers/bahendr/eigen.ps.Z*

[36] R.W.Hockney and E.A.Carmona, *Comparison of communications on the Intel iPSC/860 and Touchstone Delta*, Parallel Computing 18 (1992) 1067-72.

[37] H.Huang *A parallel algorithm for symmetric tridiagonal eigenvalue problems* CAC Document number 109 (Center for Advanced Computation, University of Illinois, 1974)

[38] E.Isaacson and H.B.Keller *Analysis of numerical methods* (Wiley 1966)

[39] S.Huss-Lederman, A.Tsao and T.Turnbull *A paralellizable eigensolver for real diagonalizable matrices with real eigenvalues* Supercomputing Research Centre Technical Report TR-91-042 (University of California, Berkeley, 1991)

[40] S.Huss-Lederman, A.Tsao and G.Zhang *A parallel implementation of the invariant subspace decomposition algorithm for dense symmetric matrices* Proc. 6th SIAM Conf. Parallel Proc. Sci. Comput. (SIAM, 1993) 367-74

[41] IBM *Parallel Engineering and Scientific Subroutine Library* Reference Guide GC23-3836-00

[42] IMSL *Library Reference Manual* IMSL Inc., Houston, Texas

[43] I.C.F.Ipsen and E.R.Jessup *Solving the symmetric tridiaconal eigenvalue problem on the hypercube* SIAM J. Sci. Stat. Computing 11 (1990) 203-29

[44] Lawson, Hanson, Kincaid and Krogh ACM Trans. Math. Software 5 (1979) 308-23

[45] C.Lanczos *An iteration method for solution of the eigenvalue problem of linear integral and differential operators* J. Res. Natl. Bureau Stand. 45 (1950) 255-82

[46] R.Lehoucq, K.Maschhoff, D.Sorensen and C.Yang *ARPACK*. Parallel ARPACK is available from *ftp://ftp.caam.rice.edu/pub/people/kristyn*

[47] R.B.Lehoucq and J.A.Scott *An evaluation of subspace iteration software for sparse nonsymmetric matrices* Technical Report, (CCLRC, 1996) also Preprint MCS-P547-1195, (Argonne National Laboratory. 1995)

[48] R.J.Littlefield and K.J.Maschhoff *Investigating the performance of parallel eigensolvers for large processor counts* Theor. Chim. Acta 84 (1993) 457-73

[49] T.Li, H.Zhang and X.Sun *Parallel homotopy algorithm for symmetric tridiagonal eigenvalue problems* SIAM J. Sci. Stat. Computing 12 (1991) 464-85

[50] S.Lo, B.Phillipe and A.Sameh *A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem* SIAM J. Sci. Stat. Computing 8 (1987) 155-65

[51] NAG *NAG Mark 15 Fortran Library Manual* NAG Ltd., Oxford, (1991)

[52] K.Maschhoff and D.Sorensen *A portable implementation of ARPACK for distributed memory parallel architectures* Preliminary proceedings, Copper Mountain Conference on Iterative Methods (1996)

[53] J.Nieplocha, R.J.Harrison, and R.J.Littlefield, *Global Arrays; A Portable Shared Memory Programming Model for Distributed Memory Computers*, Supercomputing '94 (IEEE Computer Society Press, Washington, D.C., 1994.

[54] A STLM (Spectral Transform Lanchos Method) written by a postdoc Soren Paedkjar in California IT for the Intel Delta. Contact:
Stephanie Rogers
Department of Chemistry
MC127-72
California Institute of Technology
Pasadena, CA 91125 *smrogers@cco.caltech.edu*
http://www.thefactory.com/rogers/rogers.html

[55] B.Parlett *Symmetric eigenvalue problem* (Prentice Hall 1980)

[56] C.P.Potter *A parallel divide and conquer eigensolver* EPFL Supercomputing Review 7 (1995) 27-31

[57] W.H.Press, B.P.Flannery, S.A.Teukolsky and W.T.Vetterling *Numerical Recipes – the art of scientific computing* (Cambridge University Press, 1986 and reprints)

[58] H.Pritchard, *The Global Array Tool*, CRI (13/4/95).

[59] B.T.Smith, J.M.Boyle, J.J.Dongarra, B.S.Garbow, Y.Ikebe, V.C.Klema and C.B.Moler *Matrix Eigensystem Routines – EISPACK Guide* 2nd edn. (Springer Verlag, 1976)

[60] K.Stanley *Parallel Symmetric Eigensolver Web Page* *http://www.cs.berkeley.edu/s̃tanley/symeig/index.html*

[61] J.H.Wilkinson *Algebraic eigenvalue problem* (Oxford 1988)

[62] J.H.Wilkinson and C.Reinsch *Handbook for Automatic Computation. Vol. II Linear Algebra* (Springer Verlag, 1971)