SIAM J. SCI. COMPUT. © 1999 Society for Industrial and Applied Mathematics
Vol. 20, No. 6, pp. 2223–2236

A PARALLEL DIVIDE AND CONQUER ALGORITHM FOR THE SYMMETRIC EIGENVALUE PROBLEM ON DISTRIBUTED MEMORY ARCHITECTURES*

FRANÇOISE TISSEUR† AND JACK DONGARRA‡

Abstract. We present a new parallel implementation of a divide and conquer algorithm for computing the spectral decomposition of a symmetric tridiagonal matrix on distributed memory architectures. The implementation we develop differs from other implementations in that we use a two-dimensional block cyclic distribution of the data, we use the Löwner theorem approach to compute orthogonal eigenvectors, and we introduce permutations before the back transformation of each rank-one update in order to make good use of deflation. This algorithm yields the first scalable, portable, and numerically stable parallel divide and conquer eigensolver. Numerical results confirm the effectiveness of our algorithm. We compare performance of the algorithm with that of the QR algorithm and of bisection followed by inverse iteration on an IBM SP2 and a cluster of Pentium PIIs.

Key words. divide and conquer, symmetric eigenvalue problem, tridiagonal matrix, rank-one modification, parallel algorithm, ScaLAPACK, LAPACK, distributed memory architecture

AMS subject classifications. 65F15, 68C25

PII. S1064827598336951

1. Introduction.

The divide and conquer algorithm for the symmetric tridiagonal eigenvalue problem was first developed by Cuppen [8], based on previous ideas of Golub [16] and Bunch, Nielsen, and Sorensen [5] for the solution of the secular equation. The algorithm was popularized as a practical parallel method by Dongarra and Sorensen [14], who implemented it on a shared memory machine. They concluded that divide and conquer algorithms, when properly implemented, can be many times faster than traditional ones, such as bisection followed by inverse iteration or the QR algorithm, even on serial computers. Later parallel implementations had mixed success. Using an Intel iPSC-1 hypercube, Ipsen and Jessup [22] found that their bisection implementation was more efficient than their divide and conquer implementation because of the excessive amount of data transferred between processors and unbalanced work load after the deflation process. More recently, Gates and Arbenz [15] showed that good speed-up can be achieved from distributed memory parallel implementations. However, they did not use techniques described in [18] that guarantee the orthogonality of the eigenvectors and that make good use of the deflation to speed the computation.

In this paper, we describe an efficient, scalable, and portable parallel implementation for distributed memory machines of a divide and conquer algorithm for the

*Received by the editors April 6, 1998; accepted for publication (in revised form) October 15, 1998; published electronically July 16, 1999. This work was supported in part by Oak Ridge National Laboratory, managed by Lockheed Martin Energy Research Corp. for the U.S. Department of Energy under contract DE-AC05-96OR2246 and by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office.

http://www.siam.org/journals/sisc/20-6/33695.html

†Department of Mathematics, University of Manchester, Manchester M13 9PL, England (ftisseur@ma.man.ac.uk).

‡Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, and Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831 (dongarra@cs.utk.edu).

symmetric tridiagonal eigenvalue problem. We chose to implement the rank-one update of Cuppen [8] rather than the rank-two update described in [15], [18]. We see no reason why one update should be more accurate than the other or faster in general, but Cuppen's method, as reviewed in section 2, appears to be easier to implement.

Until recently, it was thought that extended precision arithmetic was needed in the solution of the secular equation to guarantee that orthogonal eigenvectors are produced when there are close eigenvalues. However, Gu and Eisenstat [18] have found a new approach that does not require extended precision; Kahan [24] showed how to make it portable and we have used it in our implementation.

In section 3 we discuss several important issues to consider for parallel implementation of a divide and conquer algorithm, and then we derive our algorithm. We implemented our algorithm in Fortran 77 as production quality software in the ScaLAPACK model [4] and we used LAPACK divide and conquer routines [25], [27] as building blocks. The code is well suited to compute all the eigenvalues and eigenvectors of large matrices with clusters of eigenvalues. For these problems, bisection followed by inverse iteration as implemented in ScaLAPACK [4], [10] is limited by the size of the largest cluster that fits on one processor. The QR algorithm is less sensitive to the eigenvalue distribution but is more expensive in computation and communication and thus does not perform as well as the divide and conquer method. Examples that demonstrate the efficiency and numerical performance are presented in section 4.

2. Cuppen's method.

The spectral decomposition of a symmetric matrix is generally computed in three steps: tridiagonalization, diagonalization, and back transformation. Here, we consider the diagonalization $T = W\Lambda W^T$ of a symmetric tridiagonal matrix $T \in \mathbb{R}^{n\times n}$, where Λ is diagonal and W is orthogonal. Cuppen [8] introduced the decomposition

$$T = \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \rho v v^T,$$

where T_1 and T_2 differ from the corresponding submatrices of T only by their last and first diagonal coefficients, respectively. Let $T_1 = Q_1 D_1 Q_1^T, T_2 = Q_2 D_2 Q_2^T$ be spectral decompositions. Then T is orthogonally similar to the rank-one update

$$T = Q(D + \rho z z^T)Q^T, \tag{2.1}$$

where $Q = \operatorname{diag}(Q_1, Q_2)$ and $z = Q^T v$. By solving the secular equation associated with this rank-one update, we compute the spectral decomposition

$$D + \rho z z^T = U\Lambda U^T \tag{2.2}$$

and then $T = W\Lambda W^T$ with $W = QU$. A recursive application of this strategy to T_1 and T_2 leads to the divide and conquer algorithm for the symmetric tridiagonal eigenvalue problem.

Finding the spectral decomposition of the rank-one update $D + \rho z z^T$ is the heart of the divide and conquer algorithm. The eigenvalues $\{\lambda_i\}_{i=1}^n$ are the roots of the secular equation

$$f(\lambda) = 1 + \rho z^T (D - \lambda)^{-1} z, \tag{2.3}$$

and a corresponding eigenvector u is given by

$$u = (D - \lambda I)^{-1} z. \tag{2.4}$$

Each eigenvalue and corresponding eigenvector can be computed cheaply in $O(n)$ flops. Unfortunately, calculation of eigenvectors using (2.4) can lead to a loss of orthogonality for close eigenvalues. Solutions to this problem are discussed in section 3.3.

Dongarra and Sorensen [14] showed that the spectral decomposition (2.2) can potentially be reduced in size. If $z_i = 0$ for some i, then $d_i = D(i, i)$ is an eigenvalue with eigenvector the ith unit eigenvector e_i, and if there are equal d_i's, then the eigenvector basis can be rotated in order to zero out the components of z corresponding to the repeated diagonal entries. In finite precision arithmetic one needs to deflate when a z_i is nearly equal to zero and when there are nearly equal d_i's for some suitable definitions of "nearly" that ensure numerical stability is retained [14]. With suitable deflation criteria, if G is the product of all the rotations used to zero out certain components of z and if P is the accumulation of permutations used to translate the zero components of z to the bottom of z, the result is

$$PG(D + \rho zz^T)G^T P^T = \begin{pmatrix} \widetilde{D} + \rho\tilde{z}\tilde{z}^T & 0 \\ 0 & \Lambda \end{pmatrix} + E, \tag{2.5}$$

where $\|E\|_2 \le c\mathbf{u}$, with c a constant of order unity and $\mathbf{u}$ the machine precision. This deflation process is essential for the success of the divide and conquer algorithm. In practice, the dimension of $\widetilde{D}+\rho\tilde{z}\tilde{z}^T$ is usually considerably smaller than the dimension of $D+\rho zz^t$, which reduces the number of flops when computing the eigenvector matrix of T. Cuppen [8] showed that deflation is more likely to take place when the matrix is diagonally dominant When no deflation is assumed, the whole algorithm requires $\frac{4}{3}n^3 + O(n^2)$. In practice, because of deflation, it appears that the algorithm takes only $O(n^{2.3})$ flops on average and the cost can even be as low as $O(n^2)$ for some special cases (see [9]).

3. Parallelization issues and implementation details.

Divide and conquer algorithms have been successfully implemented on shared memory multiprocessors [14], [23] but difficulties have been encountered on distributed memory machines [22]. Several issues need to be addressed. A more detailed discussion of the issues discussed below can be found in [30].

3.1. Data distribution.

The first issue, and perhaps the most critical step when writing a parallel program, is how to distribute the data. Previous implementations used a one-dimensional distribution [15], [22]. Gates and Arbenz [15] used a one-dimensional row block distribution for Q, the matrix of eigenvectors, and a one-dimensional column block distribution for U, the eigenvector matrix of the rank-one updates. This distribution simplifies their parallel matrix-matrix multiplication used for the back transformation QU. However, their matrix multiplication routine grows in communication with the number of processes, making it not scalable.

By contrast, our implementation uses a two-dimensional block cyclic distribution of the matrices. The block cyclic distribution is a generalization of the block and the cyclic distributions. The processes of the parallel computer are first mapped onto a two-dimensional rectangular grid of size $P_r \times P_c$. Any general $m \times n$ dense matrix is decomposed into $m_b \times n_b$ blocks starting at its upper left corner. These blocks are then uniformly distributed in each dimension of the process grid. (See [4, Chap. 4] for more details.) We chose this data layout for several reasons. For linear algebra routines two-dimensional block cyclic distribution has been shown to be efficient and scalable [7], [20], [26]. ScaLAPACK has adopted this distribution and our aim is to write a code in the style of this software. Moreover, with this data layout, we can block partition our algorithm in order to reduce the frequency with which data is transferred between processes and thereby to reduce the fixed startup cost incurred each time a message is sent. Finally, the two-dimensional block cyclic distribution is particularly well adapted for efficient and scalable parallel matrix-matrix multiplications. As shown in section 3.4, these operations are the main computational cost of this algorithm.

3.2. Initial splitting. The second issue is the initial splitting of the work among the processes. The recursive matrix splitting leads to a hierarchy of subproblems with a data dependency graph in the form of a binary tree. If $P = P_r * P_c$ is the number of processes, this structure suggests that we distribute subproblems of dimension n/P to each of the processes. At the leaves of the tree, each process solves its subproblem independently. At each branch of the tree, the task is naturally split into two sets of processes where, in each set, processes cooperate. At the top of the tree, all processes cooperate. This is the way previous implementations have been done [15], [22]. This approach has the advantage of offering a natural parallelism for the update of the subproblems. However, Ipsen and Jessup [22] report unbalanced work load among the processes when the deflations are not evenly distributed across the sets of processes involved at the branches of the tree. In this case, the faster set of processes (those that experience deflation) will have to wait for the other set of processes before beginning the next merge. This reduces the speed-up gained though the use of the tree. A possible issue concerning load balancing the work is dynamic splitting versus static splitting [6]. In dynamic splitting, a task list is used to keep track of the various parts of the matrix during the decomposition process and to make use of data and task parallelism. This approach has been investigated[1] for the parallel implementation of the spectral divide and conquer algorithm for the unsymmetric eigenvalue problem using the matrix sign function [2]. However, we did not choose this approach, because in the symmetric case the partitioning of the matrix can be done arbitrarily and we prefer to take advantage of this opportunity.

As we use the two-dimensional block cyclic distribution, it is now natural to partition the original problem into subproblems of size n_b, that is, the size of the block distribution and not n/P as done in the previous implementations. Then, at the leaves of the tree, processes that hold a diagonal block solve their own subproblems of size $n_b \times n_b$ using the QR algorithm or the serial divide and conquer algorithm. For a grid such that $\mathrm{lcm}(P_r, P_c) = 1$, all the processes hold a subproblem at the leaves of the tree [26] and then good load balancing of the work is ensured. When $\mathrm{lcm}(P_r, P_c) = P_r$ or $\mathrm{lcm}(P_r, P_c) = P_c$ some processes hold several subproblems at the leaves of the tree and some of them hold none. However, as the computational cost of this first step is negligible compared with the computational cost of the whole algorithm, it does not matter if the work is not perfectly distributed there, as our practical experiments confirm. For a given rank-one update $Q(D + \rho zz^T)Q^T$ the processes that collaborate are those that hold a part of the global matrix Q. As we go up in the tree, a process can own data from more than one rank-one update and then has to participate in more than one parallel computation. With the two-dimensional block cyclic distribution, all the processes collaborate before the top of the tree [30]. As a consequence, work is better load balanced during the computation than in previous implementations.

3.3. Orthogonal eigenvectors. The third issue is to maintain orthogonality between eigenvectors in the presence of close eigenvalues. If $\hat{\lambda}$ is an approximate root of the secular equation (2.3) and we approximate the eigenvector u by replacing λ

[1]A ScaLAPACK prototype code is available at http://www.netlib.org/scalapack/prototype/.

in (2.4) by its approximation $\hat{\lambda}$, then when $d_j \approx \lambda$, even if $\hat{\lambda}$ is close to λ, the ratio $z_i/(d_j - \hat{\lambda})$ can be very far from the exact one. As a consequence, the computed eigenvector is very far from the true one and the resulting eigenvector matrix is far from being orthogonal. There are two approaches that solve this problem and there are trade-offs between them.

Sorensen and Tang [28] proposed using extended precision to compute the differences $d_j - \hat{\lambda}_i$. However, this approach is hard to implement portably across all the usual architectures. There are many machine-dependent tricks to make the implementation of extended precision go faster, but on some machines, such as Crays, these tricks are not valid. This is the approach used by Gates and Arbenz [15] in their implementation.

The Gu and Eisenstat approach is based on the Löwner theorem. They consider that the computed eigenvalues are the exact eigenvalues of a new rank-one update, leading to an inverse eigenvalue problem whose solution is easy to obtain. This approach can easily be implemented portably on IEEE machines and Crays using only working precision arithmetic throughout, with a trivial bit of extra work in one place to compensate for the lack of a guard digit in Cray add/subtract. This approach has been adopted for LAPACK [1], [27].

The extra precision approach is "embarrassingly" parallel, with each eigenvalue and eigenvector computed without communication, whereas the Löwner approach is not. Indeed, the Löwner approach uses a formula for the solution of the inverse eigenvalue problem that requires information about all the eigenvalues, requiring a broadcast. However, the $O(n)$ extra communication the Löwner approach uses is trivial compared with the $O(n^2)$ communication of eigenvectors elsewhere in the computation, so we chose this approach. In our implementation, we distribute the work needed for the computation of the eigenvalues and the computation of the solution of the inverse eigenvalue problem among the involved set of processes. Then solutions are broadcast over this set of processes such that each process holds the necessary information to update its own partial solution to the inverse eigenvalue problem and then to compute its local part of the eigenvector matrix U.

3.4. Back transformation. The last issue concerns the transformation of the eigenvectors of the rank-one update to the eigenvectors of the tridiagonal matrix T. This is the main cost of the divide and conquer algorithm. Let $\widetilde{D} + \rho\tilde{z}\tilde{z}^T$ be the reduced rank-one update obtained after the deflation process as defined in (2.5) and let $(\widetilde{U}, \widetilde{\Lambda})$ be its spectral decomposition. Then

$$\begin{pmatrix} \widetilde{D} + \rho\tilde{z}\tilde{z}^T & 0 \\ 0 & \bar{\Lambda} \end{pmatrix} = \begin{pmatrix} \widetilde{U} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} \widetilde{\Lambda} & 0 \\ 0 & \bar{\Lambda} \end{pmatrix} \begin{pmatrix} \widetilde{U} & 0 \\ 0 & I \end{pmatrix}^T = U\Lambda U^T,$$

and the spectral decomposition of the tridiagonal matrix T is therefore given by $T = Q(PG)^T PG(D + \rho zz^T)(PG)^T PGQ^T = W\Lambda W^T$ with $W = Q(PG)^T U$. When not properly implemented, the computation of W can be very expensive. Gu [17] suggested a permutation strategy for reorganizing the data structure of the orthogonal matrices before the back transformation. Although used in the serial LAPACK divide and conquer code, this idea has not been considered in any current parallel implementation of the divide and conquer algorithm. We derive a permutation strategy more suitable for our parallel implementation. This new strategy is one of the major contributions of our work.

To simplify the explanation of Gu's idea, we illustrate it with a 4×4 example: we suppose that $d_1 = d_3$ and that G is the Givens rotation in the (1,3) plane that zeros the third component of z. The matrix P is a permutation that moves z_3 to the bottom of z by interchanging elements 3 and 4. We indicate by "$*$" a value that has changed. Our aim is to take advantage of the block diagonal structure of $Q = \text{diag}(Q_1, Q_2)$. Note that if we apply the transformation $(PG)^T$ on the left, then the block structure of Q is preserved:

$$Q\cdot(PG)^T U = \begin{pmatrix} \times & \times & & \\ \times & \times & & \\ & & \times & \times \\ & & \times & \times \end{pmatrix} (PG)^T \begin{pmatrix} \times & \times & \times & \\ \times & \times & \times & \\ \times & \times & \times & \\ & & & 1 \end{pmatrix} = \begin{pmatrix} \times & \times & & \\ \times & \times & & \\ & & \times & \times \\ & & \times & \times \end{pmatrix} \begin{pmatrix} * & * & * & * \\ \times & \times & \times & 0 \\ * & * & * & * \\ \times & \times & \times & 0 \end{pmatrix}.$$

The product between the two last matrices is performed with 64 flops instead of the $2n^3 = 128$ flops of a full matrix product. However, if we apply $(PG)^T$ on the left, we can reduce further the number of flops:

$$Q(PG)^T \cdot U = \begin{pmatrix} * & \times & & * \\ * & \times & & * \\ * & & \times & * \\ * & & \times & * \end{pmatrix} \begin{pmatrix} \times & \times & \times & \\ \times & \times & \times & \\ \times & \times & \times & \\ & & & 1 \end{pmatrix} = \widetilde{Q}U.$$

At this step, a permutation is used to group the columns of $\widetilde{Q}$ according to their sparsity structure:

$$\widetilde{Q}U = \begin{pmatrix} * & \times & & * \\ * & \times & & * \\ * & & \times & * \\ * & & \times & * \end{pmatrix} \bar{P}\bar{P}^T \begin{pmatrix} \times & \times & \times & \\ \times & \times & \times & \\ \times & \times & \times & \\ & & & 1 \end{pmatrix} = \begin{pmatrix} \times & & * & * \\ \times & & * & * \\ & \times & * & * \\ & \times & * & * \end{pmatrix} \begin{pmatrix} * & * & * & \\ * & * & * & \\ * & * & * & \\ & & & 1 \end{pmatrix} = \bar{Q}\bar{U}.$$

Then, three matrix multiplications are performed with 48 flops involving the matrices $\bar{Q}(1{:}\,2, 1), \bar{Q}(3{:}\,4, 2), \bar{Q}(1{:}\,4, 3)$, and $\bar{U}(1{:}\,3, 1{:}\,3)$. (These are indeed matrices for larger problems.) This organization allows the BLAS to perform three matrix multiplies of minimal size.

In parallel, this strategy is hard to implement efficiently. One needs to redefine the permutation $\bar{P}$ in order to avoid communication between process columns. Let $k(q)$ be the number of deflated eigenvalues held by process column q, $0 \le q \le P_c - 1$ and $k^{'} = \min_{0\le q\le P_c-1} k(q)$. Then, in our parallel implementation, we define $\bar{P}$ so that it groups the column of Q according to their local sparsity structure and such that the resultant matrix $\bar{Q}$ has the following global structure:

$$\begin{pmatrix} \bar{Q}_{11} & \bar{Q}_{12} & 0 & \bar{Q}_{13} \\ 0 & \bar{Q}_{21} & \bar{Q}_{22} & \bar{Q}_{23} \end{pmatrix},$$

where $\bar{Q}_{11}$ contains n_1 columns of Q_1 that have not been affected by deflation, $\bar{Q}_{22}$ contains n_2 columns of Q_2 that have not been affected by deflation, $(\bar{Q}_{13}^T, \bar{Q}_{23}^T)^T$ contains $k^{'}$ columns of Q_2 that correspond to deflated eigenvalues (they are already eigenvectors of T), and $(\bar{Q}_{12}^T, \bar{Q}_{21}^T)^T$ contains the $n-(n_1+n_2+k^{'})$ remaining columns of Q. Then, for the computation of the product $\bar{Q}\bar{U}$, we use two calls to the parallel BLAS `PxGEMM` involving parts of $\bar{U}$ and the matrices $(\bar{Q}_{11}, \bar{Q}_{12}), (\bar{Q}_{21}, \bar{Q}_{22})$. Unlike in the serial implementation, we cannot assume that $k^{'} = k$, that is, that $(\bar{Q}_{13}^T, \bar{Q}_{23}^T)^T$ contains all the columns corresponding to deflated eigenvalues. This is due to the fact that $\bar{P}$ acts only on columns of Q that belong to the same process column. For a particular block size n_b, it is possible to construct an example where one process

Table 3.1
Outline of the parallel divide and conquer code.

```
      subroutine PxSTEDC( N, NB, D, E, Q, ... )
*
*     Scale the tridiagonal matrix, call PxLAED0 to solve the tridiagonal
*     eigenvalue problem, scale back when finished, sort the eigenvalues
*     and corresponding eigenvectors in ascending order with PxLASRT.


      subroutine PxLAED0( N, NB, D, E, Q, ... )
*
*     Driver of the divide and conquer code. Split the tridiagonal matrix
*     into submatrices using rank-one modification. Solve each eigenvalue
*     problem with the serial divide and conquer code xSTEDC.
*     Call PxLAED1 to merge adjacent problems.

        TSUBPBS = (N-1)/NB +1
        while (TSUBPBS > 1 )
              for i = 1:TSUBPBS/2
                  call PxLAED1(N, NB, i, TSUBPBS, D, Q, ...)
              end
        TSUBPBS = TSUBPBS / 2
        end

      subroutine PxLAED1( N, NB, i, TSUBPBS, D, Q, ...)
*     Combines eigensystems of adjacent submatrices into an eigensystem
*     for the corresponding larger matrix.
*
*     Form z=Q^Tv, the last row of Q1 and first row of Q2.
      call PxLAEDZ( N, NB, i, TSUBPBS, Q, Z, ... )
*
*     Deflate eigenvalues and permute columns of Q.
      call PxLAED2( N, NB, i TSUBPBS, K, D, Q, Z, ... )
*
*     Solution of the secular equation and computation of eigenvectors.
      call PxLAED3( N, K, D, Z, U, ... )
*
*     Back transformation.
      call PxGEMM( Q1, U1, ...)
      call PxGEMM( Q2, U2, ...)
*
```

does not deflate much and then no advantage of deflation is taken. However, we have not encountered this situation in practice and we still get good speed-up on many matrices.

3.5. The divide and conquer code. We give an outline of our parallel divide and conquer code in Table 3.1 and for more details we refer to [30]. This code uses LAPACK's serial routines whenever possible. We use the PBLAS (parallel BLAS) routine `PxGEMM` to perform our parallel matrix multiplications. This routine has a communication cost that grows with the square root of the number of processes, leading to good efficiency and scalability. All the communications are performed using the BLACS (Basic Linear Algebra Communication subprograms) [13], whose aim is to provide a portable, linear-algebra-specific layer for communication. The BLACS are available for a wide range of distributed memory machines and for both PVM (Parallel Virtual Machine) and MPI (Message Passing Interface). This makes our divide and conquer code portable on many parallel platforms, including a network of workstations supporting PVM or MPI.

4. Numerical experiments. This section concerns accuracy tests, execution times, and performance results. We compare our parallel implementation of the divide and conquer algorithm with the two parallel algorithms for solving the symmetric tridiagonal eigenvalue problem available in ScaLAPACK [4]:

- B/II: bisection followed by inverse iteration (subroutines `PxTEBZ` and `PxHEIN`). The inverse iteration algorithm can be used with two options:
  II-1: inverse iteration without a reorthogonalization process.
  II-2: inverse iteration with a reorthogonalization process when the eigenvalues are separated by less than 10^{-3} in absolute value.
- QR: the QR algorithm (subroutine `PxSTEQR2`) [10], [21].

`PxSYEVX` is the name of the expert driver[2] associated with B/II and `PxSYEV` is the simple driver associated with QR. We have written a driver called `PxSYEVD` that computes all the eigenvalues and eigenvectors of a symmetric matrix using our parallel divide and conquer routine `PxSTEDC`.

Our comparisons are based on three types of matrices. Matrix 1 has equally spaced eigenvalues from $\mathbf{u}$ to 1, matrix 2 has geometrically spaced eigenvalues from $\mathbf{u}$ to 1, and matrix 3 has clustered eigenvalues at $\mathbf{u}$. The type 3 matrices are designed to illustrate how B/II can fail to compute orthogonal eigenvectors. The tests were run on an IBM SP2 in double precision arithmetic. On this machine, $\varepsilon = 2^{-53} \approx 1.1 \times 10^{-16}$.

To verify the accuracy of our results, we measure the scaled residual error and the scaled departure from orthogonality, defined by

$$\mathcal{R} = \frac{\|A\widehat{Q} - \widehat{Q}^T\widehat{\Lambda}\|_1}{n\varepsilon\|A\|_1} \quad \text{and} \quad \mathcal{O} = \frac{\|I - \widehat{Q}^T\widehat{Q}\|_1}{n\varepsilon},$$

where $\widehat{Q}\widehat{\Lambda}\widehat{Q}^T$ is the computed spectral decomposition of A. When both quantities are small, the computed spectral decomposition is the exact spectral decomposition of a slight perturbation of the original problem. Table 4.1 shows the greatest residual and departure from orthogonality measured for matrices of type 1, 2, and 3 solved by B/II-1, B/II-2, QR, and divide and conquer. The matrices are of order $n = 1500$ with a block size $nb = 60$ on a 2×4 processor grid. For eigenvalues with equally spaced modulus, bisection followed by inverse iteration gives good numerical results and is slightly faster than the divide and conquer algorithm. This is due to the absence of communication when computing the eigenvectors, both for B/II-1 and B/II-2. However, as illustrated by matrices of type 2, if no reorthogonalization is used, numerical orthogonality can be lost with inverse iteration when the eigenvalues are poorly separated. It is clear that the reorthogonalization process greatly increases the execution time of the inverse iteration algorithm. For large clusters, the reorthogonalization process in `PxHEIN` is limited by the size of the largest cluster that fits on one processor. Unfortunately, in this case, orthogonality is not guaranteed. This phenomenon is illustrated by matrices of type 3. In the remaining experiments, we always use B/II with reorthogonalization.

We compared the relative performance of B/II, QR, and divide and conquer. In our figures, the horizontal axis is matrix dimension and the vertical axis is time divided by the time for divide and conquer, so that the divide and conquer curve is

[2] "Driver" refers to the routine that solves the eigenproblem for a full symmetric matrix by reducing the matrix to tridiagonal form, solving the tridiagonal eigenvalue problem, and transforming the eigenvectors back to those of the original matrix.

Table 4.1

Normalized residual, normalized eigenvector orthogonality, and timing for a matrix of size $n = 1500$ on an IBM-SP2 (2×4 processor grid) for bisection/inverse iteration without and with reorthogonalization, QR algorithm, and divide and conquer algorithm.

Matrix type		Eigensolvers			
		B/II-1	B/II-2	QR	DC
Uniform distribution $[\epsilon, 1]$	$\mathcal{R}$	3×10^{-4}	3×10^{-4}	3×10^{-4}	2×10^{-4}
	$\mathcal{O}$	0.20	0.17	0.55	0.27
	Time	52	52	120	58
Geometrical distribution $[\epsilon, 1]$	$\mathcal{R}$	3×10^{-4}	3×10^{-4}	5×10^{-4}	4×10^{-4}
	$\mathcal{O}$	$\geq 1,000$	88.03	0.23	0.20
	Time	53	137	95	51
Clustered at ε	$\mathcal{R}$	4×10^{-4}	4×10^{-4}	4×10^{-4}	4×10^{-4}
	$\mathcal{O}$	$\geq 1,000$	$\geq 1,000$	0.50	0.16
	Time	52	139	120	47

constant at 1. It is clear from Figures 4.1 and 4.2, which correspond to the spectral decomposition of the tridiagonal matrix T and the symmetric matrix A, respectively, that divide and conquer competes with bisection followed by inverse iteration when the eigenvalues of the matrix are well separated. For inverse iteration, this eigenvalue distribution is good since no reorthogonalization of eigenvectors is required. For divide and conquer it is bad since this means there is little deflation within intermediate problems. Note that the execution times of QR are much larger. This distinction in speed between QR or B/II and divide and conquer is more noticeable in Figure 4.1 (speed-up up to 6.5) than in Figure 4.2 (speed-up up to 2) because Figure 4.2 includes the overhead of the tridiagonalization and back transformation processes. As illustrated in Figure 4.3, divide and conquer runs much faster than B/II as soon as eigenvalues are poorly separated or in clusters. We also compare execution times of the tridiagonalization, QR, B/II-2, and back transformation relative to the execution time of divide and conquer. From Figure 4.4, it appears that when using the QR algorithm for computing all the eigenvalues and eigenvectors of a symmetric matrix, the bottleneck is the spectral decomposition of the tridiagonal matrix. This is not true any more when using our parallel divide and conquer algorithm: spectral decomposition of the tridiagonal matrix is now faster than the tridiagonalization and back transformation of the eigenvectors. Effort as in [19] should be made to improve the tridiagonalization and back transformation.

We measured the performances of `PDSTEDC` on an IBM SP2 (Figure 4.5) and on a cluster of 300 MHz Intel PII processors using a 100 Mbit Switch Ethernet connection (Figure 4.6). In our figures, the horizontal axis is the number of processors and the vertical axis is the number of flops per second obtained when the size of the problem is maintained constant on each process. The performance increases with the number of processors, which illustrates the scalability of our parallel implementation. These measures have been done using the Level 3 BLAS of ATLAS (Automatically Tuned Linear Algebra Software) [31], which runs at a peak of 440 Mflop/s on the SP2 and 190 Mflop/s on the PII. Our code runs at 50% of the peak performance of matrix multiplication on the SP2 and 40% of the corresponding peak on the cluster of P/II. Note that these percentages take into account the time spent at the end of the computation to sort the eigenvalues and corresponding eigenvectors into increasing order.

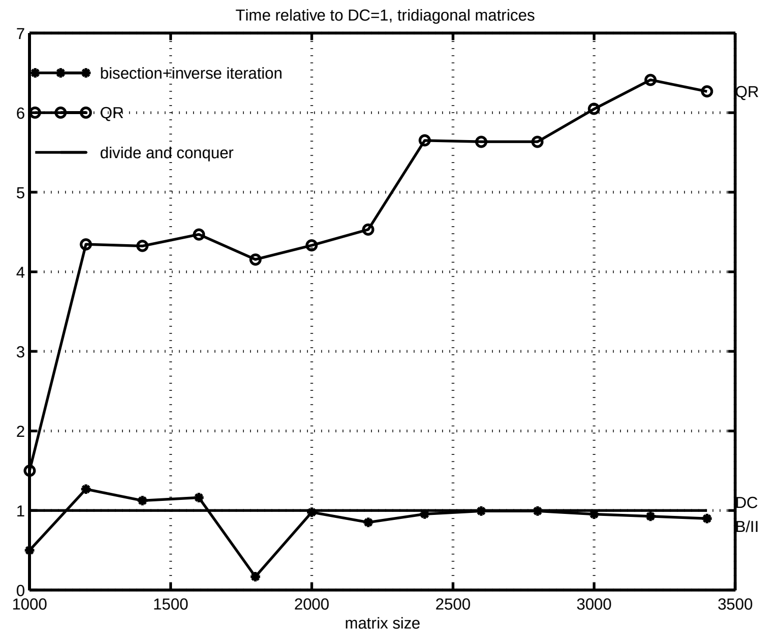

FIG. 4.1. *Execution times of* PDTEBZ+PDSTEIN *(B/II) and* PDSTEQR2 *(QR) relative to* PDSTEDC *(DC), on an IBM SP2, using* 8 *nodes. Tridiagonal matrices, eigenvalues of equally spaced modulus.*

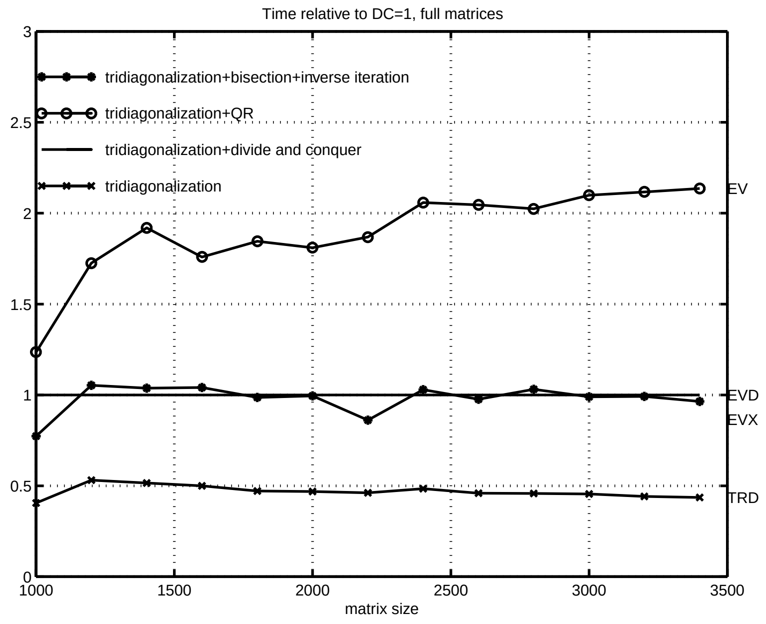

FIG. 4.2. *Execution times of* PDSYEVX *(B/II),* PDSYEV *(QR), and* PDSYTRD *(tridiagonalization) relative to* PDSYEVD *(DC), on an IBM SP2, using* 8 *nodes. Full matrices, eigenvalues of equally spaced modulus.*

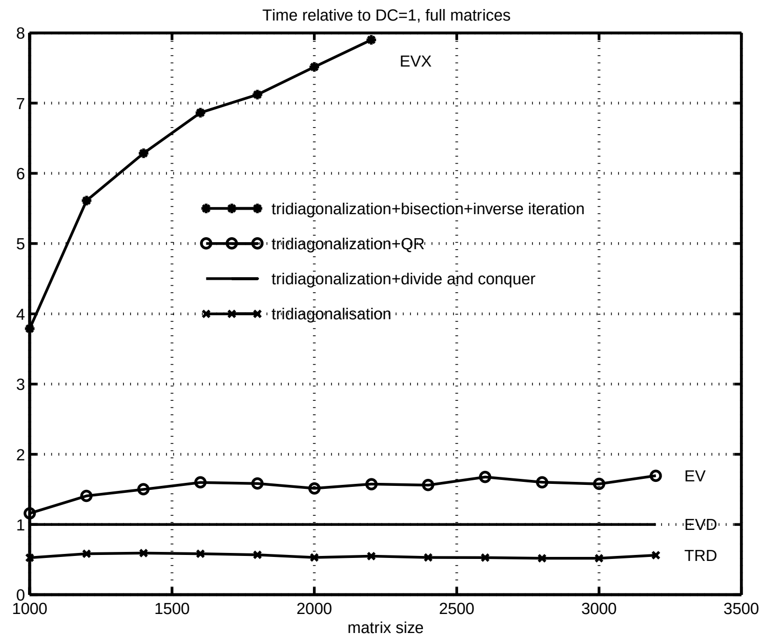

FIG. 4.3. *Execution times of* PDSYEVX *(B/II),* PDSYEV *(QR), and* PDSYTRD *(tridiagonalization) relative to* PDSYEVD *(DC), on an IBM SP2, using* 8 *nodes. Full matrices, eigenvalues of geometrically spaced modulus.*

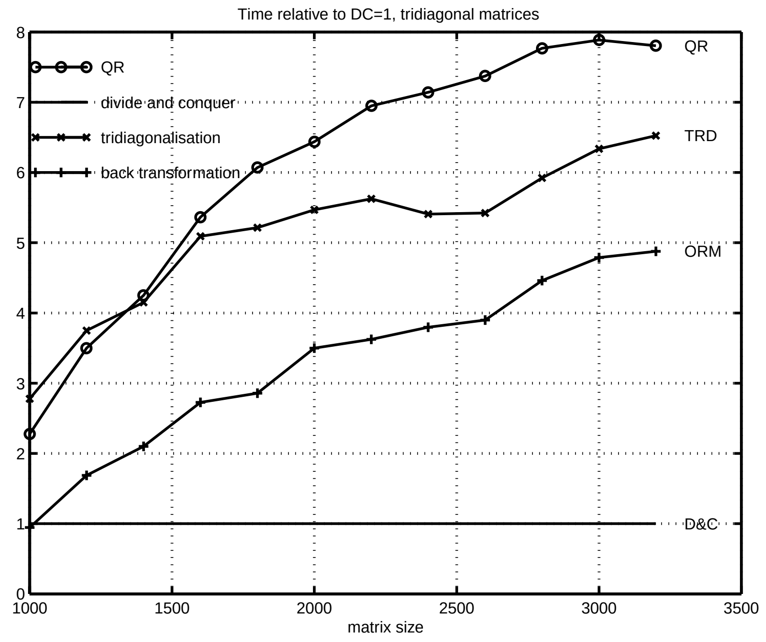

FIG. 4.4. *Execution times of* PDSTEQR2 *(QR),* PDSYTRD *(tridiagonalization), and* PDORMTR *(back transformation) relative to* PDSTEDC *(DC). Measured on an IBM SP2, using* 8 *nodes. Tridiagonal matrices, eigenvalues of geometrically spaced modulus.*

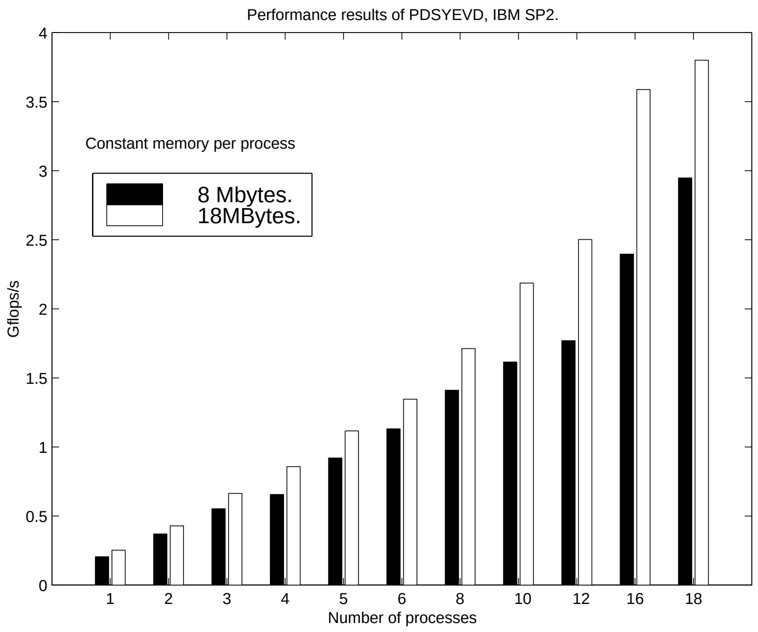

Fig. 4.5. *Performance of* PDSTEDC, *IBM SP*2.

5. Conclusions.

For serial and shared memory machines, divide and conquer is one of the fastest available algorithms for finding all the eigenvalues and eigenvectors of a large dense symmetric matrix. By contrast, implementations of this algorithm on distributed memory machines have in the past posed difficulties.

In this paper, we showed that divide and conquer can be efficiently parallelized on distributed memory machines. By using the Löwner theorem approach, good numerical eigendecompositions are obtained in all situations. From the point of view of execution time, our results seem to be better for most cases when compared with the parallel execution time of QR and bisection followed by inverse iteration available in the ScaLAPACK library. Performance results on the IBM SP2 and a cluster of PC PIIs demonstrate the scalability and portability of our algorithm. Good efficiency is mainly obtained by exploiting the data parallelism inherent to this algorithm rather than its task parallelism. For this, we concentrated our efforts on a good implementation of the back transformation process in order to reach maximum speed-up for the matrix multiplications. Unlike in previous implementations, the number of processes is not required to be a power of two. This implementation will be incorporated in the ScaLAPACK library.

Recent work [11] has been done on an algorithm based on inverse iteration which may provide a faster and more accurate algorithm and should also yield an embarrassingly parallel algorithm. Unfortunately, there is no parallel implementation available at this time, so we could not compare this new method with divide and conquer.

We showed that in contrast to the ScaLAPACK QR algorithm implementation, the spectral decomposition of the tridiagonal matrix is no longer the bottleneck. Efforts as in [19] should be made to improve the tridiagonalization and the back transformation of the eigenvector matrix of the tridiagonal form to the original one.

The main limitation of this proposed parallel algorithm is the amount of storage

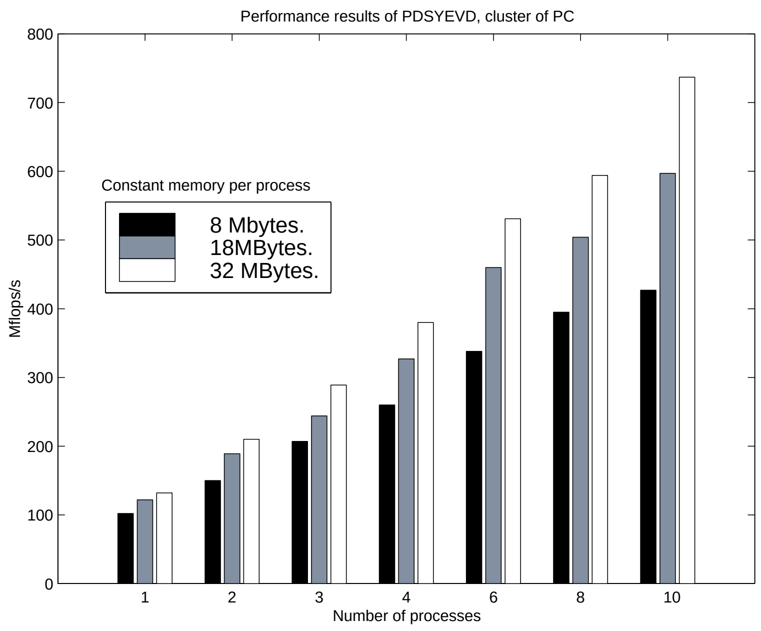

FIG. 4.6. *Performance of* PDSEDC*, cluster of* 300*-MHz Intel PII processors using a* 100*-Mbit Switch Ethernet connection.*

needed. Compared with the ScaLAPACK QR implementation, $2n^2$ extra storage locations are required to perform the back transformation in the last step of the divide and conquer algorithm. This is the price we pay for using Level 3 BLAS operations. It is worth noting that in most of the cases, not all this storage is used, because of deflation. Unfortunately, ideas as developed in [29] for the sequential divide and conquer seem hard to implement efficiently in parallel as they require a lot of communication. As in many algorithms, there is a trade off between good efficiency and workspace [3], [12]. Such a trade-off appears also in parallel implementations of inverse iteration when reorthogonalization of the eigenvectors is performed.

In future work, the authors plan to use ideas developed in this paper for the development of a parallel implementation of the divide and conquer algorithm for the singular value decomposition.

REFERENCES

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, SIAM, 2nd ed., Philadelphia, PA, 1995.

[2] Z. Bai, J. W. Demmel, J. J. Dongarra, A. Petitet, H. Robinson, and K. Stanley, *The spectral decomposition of nonsymmetric matrices on distributed memory parallel computers*, SIAM J. Sci. Comput., 18 (1997), pp. 1446–1461.

[3] C. Bischof, S. Huss-Lederman, X. Sun, A. Tsao, and T. Turnbull, *Parallel performance of a symmetric eigensolver based on the invariant subspace decomposition approach*, in Proceedings of Scalable High Performance Computing Conference '94, Knoxville, TN, 1994, pp. 32–39. PRISM Working Note 15.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, SIAM, Philadelphia, PA, 1997.

[5] J. R. Bunch, C. P. Nielsen, and D. C. Sorensen, *Rank-one modification of the symmetric eigenproblem*, Numer. Math., 31 (1978), pp. 31–48.

[6] S. Chakrabarti, J. Demmel, and K. Yelick, *Modeling the Benefits of Mixed Data and Task Parallelism*, Technical Report CS-95-289, Department of Computer Science, University of Tennessee, Knoxville, TN, 1995. LAPACK Working Note 97.

[7] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, *ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers*, Technical Report CS-92-181, Department of Computer Science, University of Tennessee, Knoxville, TN, 1992. LAPACK Working Note 55.

[8] J. J. M. Cuppen, *A divide and conquer method for the symmetric tridiagonal eigenproblem*, Numer. Math., 36 (1981), pp. 177–195.

[9] J. W. Demmel, *Applied Numerical Linear Algebra*, SIAM, Philadelphia, PA, 1997.

[10] J. W. Demmel and K. Stanley, *The Performance of Finding Eigenvalues and Eigenvectors of Dense Symmetric Matrices on Distributed Memory Computers*, Technical Report CS-94-254, Department of Computer Science, University of Tennessee, Knoxville, TN, 1994. LAPACK Working Note 86.

[11] I. Dhillon, *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*, Ph.D. thesis, University of California, Berkeley, CA, 1997.

[12] S. Domas and F. Tisseur, *Parallel implementation of a symmetric eigensolver based on the Yau and Lu method*, in Vector and Parallel Processing—VECPAR'96, Lecture Notes in Comput. Sci. 1215, Springer-Verlag, Berlin, 1997, pp. 140–153.

[13] J. Dongarra and R. Whaley, *A User's Guide to the BLACS v*1.1, Technical Report CS-97-281, Department of Computer Science, University of Tennessee, Knoxville, TN, 1997. LAPACK Working Note 94.

[14] J. Dongarra and D. Sorensen, *A fully parallel algorithm for the symmetric eigenvalue problem*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. s139–s154.

[15] K. Gates and P. Arbenz, *Parallel Divide and Conquer Algorithms for the Symmetric Tridiagonal Eigenproblem*, Technical Report, Institute for Scientific Computing, ETH Zurich, Zurich, Switzerland, 1994.

[16] G. H. Golub, *Some modified matrix eigenvalue problems*, SIAM Rev., 15 (1973), pp. 318–334.

[17] M. Gu, *Studies in Numerical Linear Algebra*, Ph.D. thesis, Yale University, New Haven, CT, 1993.

[18] M. Gu and S. C. Eisenstat, *A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem*, SIAM J. Matrix Anal. Appl., 16 (1995), pp. 172–191.

[19] B. Hendrickson, E. Jessup, and C. Smith, *Toward an efficient parallel eigensolver for dense symmetric matrices*, SIAM J. Sci. Comput., 20 (1999), pp. 1132–1154.

[20] B. Hendrickson and D. Womble, *The torus-wrap mapping for dense matrix calculations on massively parallel computers*, SIAM J. Sci. Comput., 15 (1994), pp. 1201–1226.

[21] G. Henry, *Improving Data Re-Use in Eigenvalue-Related Computations*, Ph.D. thesis, Cornell University, Ithica, NY, 1994.

[22] I. C. F. Ipsen and E. R. Jessup, *Solving the symmetric tridiagonal eigenvalue problem on the hypercube*, SIAM J. Sci. Statist. Comput., 11 (1990), pp. 203–229.

[23] E. R. Jessup and D. C. Sorensen, *A parallel algorithm for computing the singular value decomposition of a matrix*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 530–548.

[24] W. Kahan, *Rank-*1 *Perturbed Diagonal's Eigensystem*, unpublished manuscript, 1989.

[25] R.-C. Li, *Solving the secular equation stably and efficiently*, Technical Report, Department of Mathematics, University of California, Berkeley, CA, 1993. LAPACK Working Note 89.

[26] A. Petitet, *Algorithmic Redistribution Methods for Block Cyclic Decompositions*, Ph.D. thesis, University of Tennessee, Knoxville, TN, 1996.

[27] J. Rutter, *A Serial Implementation of Cuppen's Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem*, Technical Report CS-94-225, Department of Computer Science, University of Tennessee, Knoxville, TN, 1994. LAPACK Working Note 69.

[28] D. C. Sorensen and P. T. P. Tang, *On the orthogonality of eigenvectors computed by divide-and-conquer techniques*, SIAM J. Numer. Anal., 28 (1991), pp. 1752–1775.

[29] F. Tisseur, *Workspace Reduction for the Divide and Conquer Algorithm*, manuscript, 1997.

[30] F. Tisseur and J. J. Dongarra, *Parallelizing the Divide and Conquer Algorithm for the Symmetric Tridiagonal Eigenvalue Problem on Distributed Memory Architectures*, Technical Report CS-98-381, Department of Computer Science, University of Tennessee, Knoxville, TN, 1998. LAPACK Working Note 132.

[31] R. C. Whaley and J. Dongarra, *Automatically Tuned Linear Algebra Software*, Technical Report CS-97-366, Department of Computer Science, University of Tennessee, Knoxville, TN, 1997. LAPACK Working Note 131.