ISSN 1360-1725

THE UNIVERSITY of MANCHESTER

UMIST

Parallelizing the Divide and Conquer Algorithm for the Symmetric Tridiagonal Eigenvalue Problem on Distributed Memory Architectures

F. Tisseur and J. Dongarra

Numerical Analysis Report No. 317

March 1998

Manchester Centre for Computational Mathematics
Numerical Analysis Reports

DEPARTMENTS OF MATHEMATICS

Reports available from:
Department of Mathematics
University of Manchester
Manchester M13 9PL
England

And over the World-Wide Web from URLs
http://www.ma.man.ac.uk/MCCM/MCCM.html
ftp://ftp.ma.man.ac.uk/pub/narep

Parallelizing the Divide and Conquer Algorithm for the Symmetric Tridiagonal Eigenvalue Problem on Distributed Memory Architectures *

Françoise Tisseur† Jack Dongarra‡

February 24, 1998

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 PII's.

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

*This work supported in part by Oak Ridge National Laboratory, managed by Lockheed Martin Energy Research Corp. for the U.S. Department of Energy under contract number DE-AC05-96OR2246 and by the Defense Advanced Research Projects Agency under contract DAAL03-91-C-0047, administered by the Army Research Office.

†Department of Mathematics, University of Manchester, Manchester, M13 9PL, England (`ftisseur@ma.man.ac.uk`, `http://www.ma.man.ac.uk/~ftisseur/`).

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

AMS subject classifications. 65F15, 68C25

1 Introduction

The divide and conquer algorithm is an important recent development for solving the tridiagonal symmetric eigenvalue problem. The algorithm was first developed by Cuppen [8] based on previous ideas of Golub [17] and Bunch, Nielson and Sorensen [5] for the solution of the secular equation and made popular as a practical parallel method by Dongarra and Sorensen [14]. This simple and attractive algorithm was considered unstable for a while because of a lack of orthogonality in the computed eigenvectors. It was thought that extended precision arithmetic was needed in the solution of the secular equation to guarantee that sufficiently orthogonal eigenvectors are produced when there are close eigenvalues. Recently, however, Gu and Eisenstat [20] have found a new approach that does not require extended precision and we have used it in our implementation.

The divide and conquer algorithm has natural parallelism as the initial problem is partitioned into several subproblems that can be solved independently. Early parallel implementations had mixed success. Dongarra and Sorensen [14] and later Darbyshire [9] wrote an implementation for shared memory machines (Alliant FX/8, KSR1). 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. Hence, a version has been incorporated in LAPACK [1]. Ipsen and Jessup [23] compared their parallel implementations of the divide and conquer algorithm and the bisection algorithm on the Intel iPSC-1 hypercube. They found that their bisection implementation was more efficient than their divide and conquer implementation because of the excessive amount of data that was transferred between processors and also because of unbalanced work load after the deflation process. More recently, Gates and Arbenz [16] with an implementation for the Intel Paragon and Fachin [15] with an implementation on a network of T800 transputers showed that good speed-up can be achieved from distributed memory parallel implementations. However, their implementations are not as efficient as they could have been. They did not use techniques described in [20] that guarantee the orthogonality of the eigenvectors and that make good use of the deflation in order 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 symmetric tridiagonal eigenvalue problem.

Divide and conquer methods consist of an initial partition of the problem into subproblems and then, after some appropriate computations done on these individual subproblems, results are joined together using rank-r updates ($r > 1$). We chose to implement the rank-one update of Cuppen [8] rather than the rank-two update used in [16], [20]. A priori, 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.

In Section 3 we discuss several important issues to consider for parallel implementations of a divide and conquer algorithm. Then, in Section 4, we derive our algorithm. We have implemented our algorithm in Fortran 77 as production quality software in the ScaLAPACK model [4]. Our algorithm 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], [11] 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 6.

2 Cuppen's Method

Solving the symmetric eigenvalue problem consists, in general, of three steps: the symmetric matrix A is reduced to tridiagonal form T, then one computes the eigenvalues and eigenvectors of T and finally, one computes the eigenvectors of A from the eigenvectors of T.

In this section we consider the problem of determining the spectral decomposition

$$T = W \Lambda W^T$$

of a symmetric tridiagonal matrix $T \in \mathbb{R}^{n \times n}$, where Λ is diagonal and W is orthogonal.

Cuppen [8] divides the original problem into subproblems of smaller size by introducing the decomposition:

$$T = \begin{array}{cc} & \begin{array}{cc} \scriptstyle k & \scriptstyle n-k \end{array} \\ \begin{array}{c} \scriptstyle k \\ \scriptstyle n-k \end{array} & \begin{pmatrix} T_1 & \beta e_k e_1^T \\ \beta e_1 e_k^T & T_2 \end{pmatrix} \end{array} = \begin{pmatrix} \widehat{T}_1 & 0 \\ 0 & \widehat{T}_2 \end{pmatrix} + \theta\beta \begin{pmatrix} e_k \\ \theta^{-1} e_1 \end{pmatrix} \begin{pmatrix} e_k^T & \theta^{-1} e_1^T \end{pmatrix}$$

where $1 \leq k \leq n$, e_j represents the jth canonical vector of appropriate dimension, β is the kth off-diagonal element of T, and $\widehat{T}_1$ and $\widehat{T}_2$ differ from the corresponding submatrices of T only by their last and first diagonal coefficient, respectively. This is the *divide phase*. Dongarra and Sorensen [14] introduced the factor θ to avoid cancellation when forming the new diagonal elements of $\operatorname{diag}(\widehat{T}_1, \widehat{T}_2)$. We now have two independent symmetric tridiagonal eigenvalue problems of order k and $n-k$. Let

$$\widehat{T}_1 = Q_1 D_1 Q_1^T, \quad \widehat{T}_2 = Q_2 D_2 Q_2^T \tag{2.1}$$

be their spectral decompositions and

$$z = \begin{pmatrix} Q_1^T & 0 \\ 0 & Q_2^T \end{pmatrix} \begin{pmatrix} e_k \\ \theta^{-1} e_1 \end{pmatrix}. \tag{2.2}$$

Then we get

$$\begin{aligned} T &= \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix} \left\{ \begin{pmatrix} D_1 & 0 \\ 0 & D_2 \end{pmatrix} + \rho z z^T \right\} \begin{pmatrix} Q_1 & 0 \\ 0 & Q_2 \end{pmatrix}^T, \ \rho = \theta\beta, \qquad (2.3) \\ &= Q(D + \rho z z^T) Q^T \end{aligned}$$

and the eigenvalues of T are therefore those of $D + \rho z z^T$. Finding the spectral decomposition

$$D + \rho z z^T = U \Lambda U^T$$

of a rank-one update of a diagonal matrix is the heart of the divide and conquer algorithm. This is the *conquer phase*. Then it follows that

$$T = W \Lambda W^T \quad \text{with } W = QU. \tag{2.4}$$

A recursive application of the strategy described above on the two tridiagonal matrices in (2.1) leads to the divide and conquer method for the symmetric tridiagonal eigenvalue problem.

2.1 Computing the Spectral Decomposition of a Rank-One Perturbed Matrix

An updating technique as described in [5], [8], [17] can be used to compute the spectral decomposition of a rank-one perturbed matrix

$$D + \rho z z^T = U \Lambda U^T \tag{2.5}$$

where $D = \operatorname{diag}(d_1, d_2, \ldots, d_n)$, $z = (z_1, z_2, \ldots, z_n)$ and ρ is a nonzero scalar. By setting the characteristic polynomial of $D + \rho z z^T$ equal to zero, we find that the eigenvalues $\{\lambda_i\}_{i=1}^n$ of $D + \rho z z^T$ are the roots of

$$f(\lambda) = 1 + \rho \sum_{i=1}^{n} \frac{z_i^2}{d_i - \lambda}, \tag{2.6}$$

which is called the *secular equation.*

Many methods have been suggested for solving the secular equation (see Melman [27] for a survey). Each eigenvalue is computed in $O(n)$ flops. A corresponding normalized eigenvector u can be computed from the formula

$$u = \frac{(D-\lambda I)^{-1}z}{||(D-\lambda I)^{-1}z||} = \left(\frac{z_1}{d_1-\lambda}, \dots, \frac{z_n}{d_n-\lambda}\right) \Bigg/ \sqrt{\sum_{j=1}^{n} \frac{z_j^2}{(d_j-\lambda)^2}} \tag{2.7}$$

in only $O(n)$ flops. Thus, the spectral decomposition of a rank-one perturbed matrix can be computed in $O(n^2)$ flops. Unfortunately, calculation of the eigenvectors using (2.7) can lead to a loss of orthogonality for close eigenvalues. We discuss solutions to this problem in Section 4.2.

2.2 Deflation

Dongarra and Sorensen [14] showed that the problem (2.5) can potentially be reduced in size. If $z_i = 0$ for some i, we see from (2.3) that d_i is an eigenvalue of $D + \rho zz^T$ with eigenvector e_i. Moreover, if D has an eigenvalue d_i of multiplicity $m > 1$, we can rotate the eigenvector basis in order to zero out the component of z corresponding to the repeated eigenvalues. Then, we can remove rows and columns from $D + \rho zz^T$ corresponding to zero components of z.

When working in finite precision arithmetic, we must deal with the problem of the z_i's nearly equal to zero and nearly equal d_i's. Then, in order to describe precisely when we can deflate, we need to define a tolerance η. Let $\eta = \varepsilon \|D+\rho zz^T\|_2$ where ε is the machine precision.

We say that the first type of deflation arises when $|\rho z_i| \leq \eta$. Now assume that $\|z\|_2 = 1$. We have

$$\|(D + \rho zz^T)e_i - d_i e_i\|_2 = |\rho z_i| \|z\|_2 \leq \eta.$$

Then (d_i, e_i) may be considered as an approximate eigenpair for $D + \rho zz^T$ and z_i is set to zero.

The second type of deflation comes from a Givens rotation applied to $D+\rho zz^T$ in order to set a component of z equal to zero. Suppose that $|z_i z_j||d_i - d_j|/\sqrt{z_i^2 + z_j^2} \leq \eta$. Let G_{ij} be the Givens rotation defined by

$$[e_i, e_j]^T G_{ij} [e_i, e_j] = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

with $c = z_i/r,\ s = z_j/r,\ r = \sqrt{z_i^2 + z_j^2}$ and $c^2 + s^2 = 1$. Then

$$G_{ij}(D + \rho zz^T)G_{ij}^T = \widetilde{D} + \rho \tilde{z}\tilde{z}^T + E_{ij}, \quad \|E_{ij}\|_2 \leq \eta,$$

where $\tilde{z}_i = r^2, \tilde{z}_j = 0, \tilde{d}_i = d_i c^2 + d_j s^2, \tilde{d}_j = d_i s^2 + d_j c^2$ and $\|E_{ij}\|_2 = |(d_i - d_j)cs|$.

The result of recognizing all these deflations is to replace the rank-one update problem $D + \rho z z^T$ with one of smaller size. Hence, 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

$$PG(D + \rho z z^T)G^T P^T = \begin{pmatrix} \widetilde{D} + \rho \tilde{z}\tilde{z}^T & 0 \\ 0 & \Lambda \end{pmatrix} + E, \qquad \|E\|_2 \le c\eta$$

with c a constant of order unity. The matrix $\widetilde{D} + \rho \tilde{z}\tilde{z}^T$ has only simple eigenvalues and all the elements of $\tilde{z}$ are nonzero.

The 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 $D + \rho z z^T$ reducing the number of flops when computing the eigenvector matrix of T in (2.4). Cuppen [8] showed that deflation are more likely to take place when the matrix is diagonally dominant.

2.3 Algorithm and Complexity

The divide and conquer algorithm is naturally expressed in recursive form as follows.

Procedure dc_eigendecomposition(T, W, Λ)

* *From input T compute output W, Λ such that $T = W\Lambda W^T$.* *\

  if *T is 1-by-1*

    return $Q = 1, \quad \Lambda = T$

  else

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

    call dc_eigendecomposition(T_1, W_1, Λ_1)

    call dc_eigendecomposition(T_2, W_2, Λ_2)

    Form $D + \rho z z^T$ *from* $\Lambda_1, W_1, \Lambda_2, W_2$

    Find eigenvalues Λ *and eigenvectors* U *of* $D + \rho z z^T$

    Form $W = \begin{pmatrix} W_1 & 0 \\ 0 & W_2 \end{pmatrix} U$

    return W, Λ

  end

The recursion can be carried on until we reach a 2×2 or 1×1 eigenvalue problem or it can be terminated with an $n_0 \times n_0$ problem and we can use the QR algorithm or some other method to solve the tridiagonal problem. Note that we have not

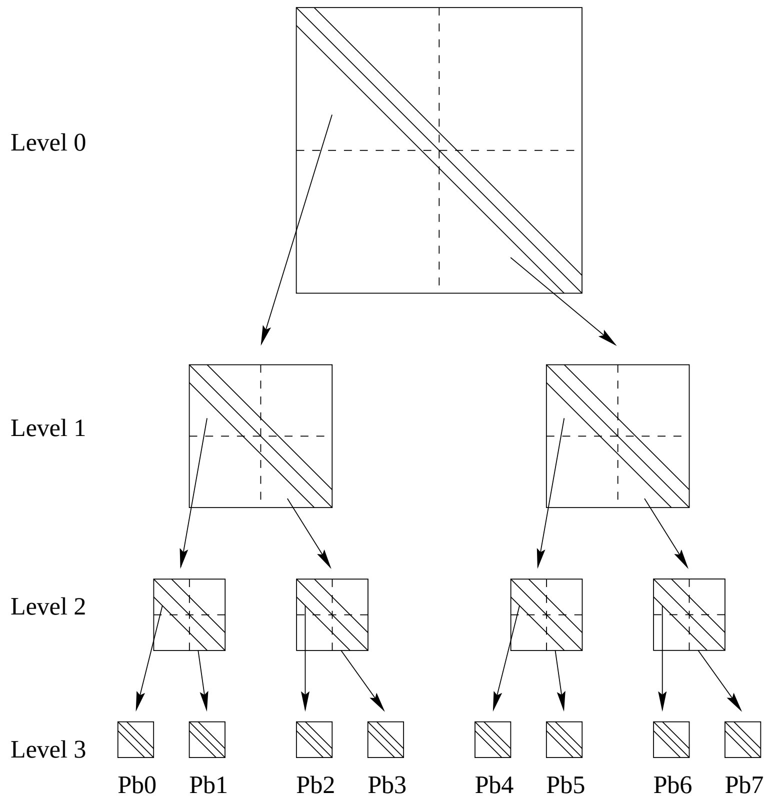

Figure 3.1: A divide and conquer tree.

specified the dimensions of T_1 and T_2. We refer to Section 4.1 for details of how we choose the size of the subproblems in our parallel implementation.

We count as one flop, an elementary floating point operation $+, -, /$ or $*$. Assuming no deflation and ignoring the terms in $O(n^2)$, the number of flops $t(n)$ to run dc_eigendecomposition for an $n \times n$ tridiagonal matrix$T$ satisfies the recursion

$$t(n) = n^3 + 2t(n/2),$$

which has the solution

$$t(n) \approx \frac{4}{3}n^3 + O(n^2).$$

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 [10]).

3 Parallelization Issues

Divide and conquer algorithms have been successfully implemented on shared memory multiprocessors for solving the symmetric tridiagonal eigenvalue problem and for the computation of the singular value decomposition of bidiagonal matrices [14],

[24]. By contrast, the implementation of these algorithms on distributed memory machines poses difficulties. Several issues need to be addressed and several implementations are possible.

The first issue is how to split the work among the processors. As shown in Figure 3.1, the recursive matrix splitting leads to a hierarchy of subproblems with a data dependency graph in the form of a binary tree. This structure suggests a natural way to split the work among the processes. If P is the number of processes, the smallest subproblems are chosen to be of dimension n/P, lying at the leaves of the tree. At the top of the tree, all processes cooperate. At each branch of the tree, the task is naturally split in two sets of processes where, in each set, processes cooperate. At the leaves of the tree, each process solves its subproblem independently. This is the way previous implementations have been done [15], [16], [23]. This approach offers a natural parallelism for the update of the subproblems. Ipsen and Jessup [23] 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 speedup gained though the use of the tree. Gates and Arbenz [16] showed that if we assume that the work corresponding to any node in the tree is well balanced among the processors, then the implementation should still achieve 85% efficiency even in the worst case of bad distribution of deflations. However, it is worth considering this problem for our implementation. A possible issue is dynamic splitting versus static splitting [6]. A task list is used to keep track of the various parts of the matrix during the decomposition process and 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]. 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. By contrast with previous implementations we use a splitting different from the natural splitting associated with the binary tree. We explain our approach in Section 4.1.

The second issue is to maintain orthogonality between eigenvectors in the presence of close eigenvalues. There are two approaches: the extra precision approach of Sorensen and Tang [30], used by Gates and Arbenz [16] in their implementation, and the Löwner Theorem approach proposed by Gu and Eisenstat [19] and adopted for LAPACK [1], [29]. There are trade-offs that we shall discuss in Section 4.2 between these two approaches.

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

The third issue is the back transformation process, which is of great importance for the success of divide and conquer algorithms because it reduces the cost of forming the eigenvector matrix of the tridiagonal form. We explain in Section 4.3 the idea of Gu and Eisenstat for reorganizing the data structure of the orthogonal matrices before the back transformation and we propose a parallel implementation of this approach. While used in the serial LAPACK divide and conquer code, this idea has never been considered in any current parallel implementation of the divide and conquer algorithm.

The last 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 [16], [23]. Gates and Arbenz [16] 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 data in the style of ScaLAPACK. 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.

4 Implementation Details

We now describe a parallel implementation of the divide and conquer algorithm that addresses the issues discussed in the previous section. Our goal was to write an efficient, reliable, portable and scalable code that follows the conventions of the ScaLAPACK software [4].

On shared-memory concurrent computers, LAPACK seeks to make efficient use of the memory hierarchy by maximizing data reuse. Specifically, LAPACK casts linear algebra computations in terms of block-oriented matrix-matrix operations whenever possible, enabling maximal use of Level 3 BLAS (matrix-matrix operations). An analogous approach has been taken by ScaLAPACK for distributed-memory machines. ScaLAPACK uses block partitioned algorithms in order to reduce the frequency with which data must be transferred between processes and thereby to reduce the fixed startup cost incurred each time a message is sent.

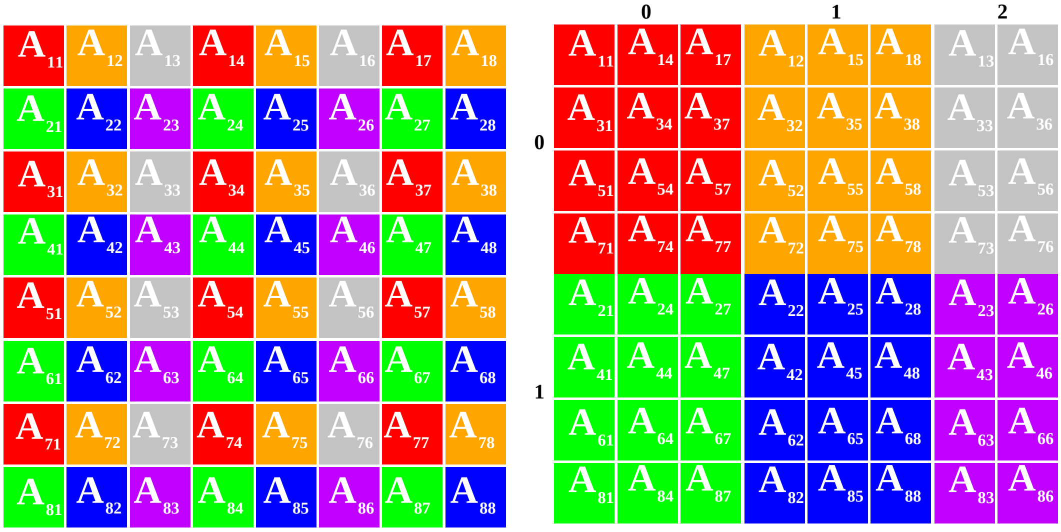

Figure 4.1: Global (left) and the distributed (right) views of the matrix ($P = 2, Q = 3$).

4.1 Data Distribution

In contrast to all previous parallel implementations for distributed memory machines [15], [16], [23], we use a two-dimensional block cyclic distribution for U, the eigenvector matrix of the rank-one update, and Q, the matrix of the back transformation. For linear algebra routines and matrix multiplications, two-dimensional block cyclic distribution has been shown to be efficient and scalable [7],[21], [28]. The ScaLAPACK software has adopted this data layout.

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 $P \times Q$. 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 $P \times Q$ process grid as illustrated in Figure 4.1. In our implementation, we impose the block size to be equal in each direction: $m_b = n_b$.

Previous parallel implementations gave a subproblem of size $n^2/(Q \times P)$ to each processor. As we use the two-dimensional block cyclic distribution, it is now natural to partition the original problem into subproblems of size n_b. 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. Some processes may hold several subproblems and some of them none. 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 well distributed there. However, good load balancing of the work is assured when the grid $P \times Q$ of processes is such that $\mathrm{lcm}(P, Q) = 1$. In this case, at the leaves of the tree, all the processes hold a subproblem [28]. The worst case happens when $\mathrm{lcm}(P, Q) = P$ or $\mathrm{lcm}(P, Q) = Q$.

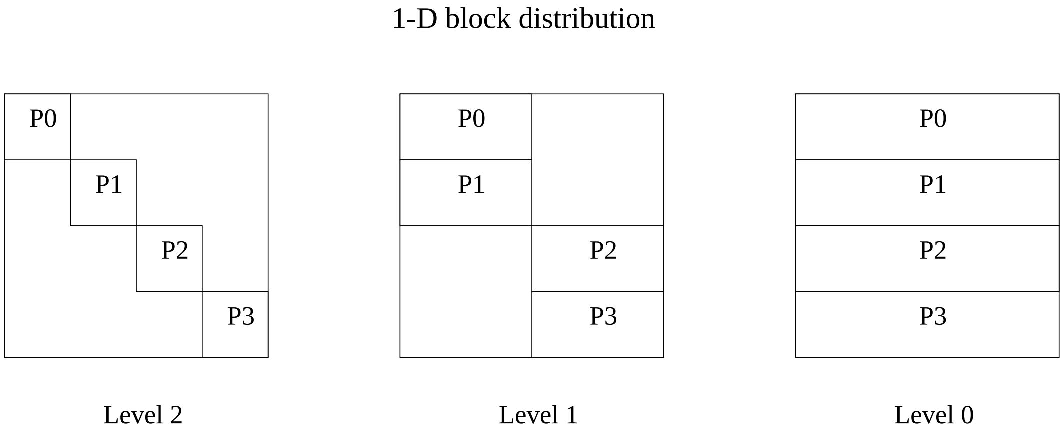

Figure 4.2: Active part of the matrix Q held by each process.

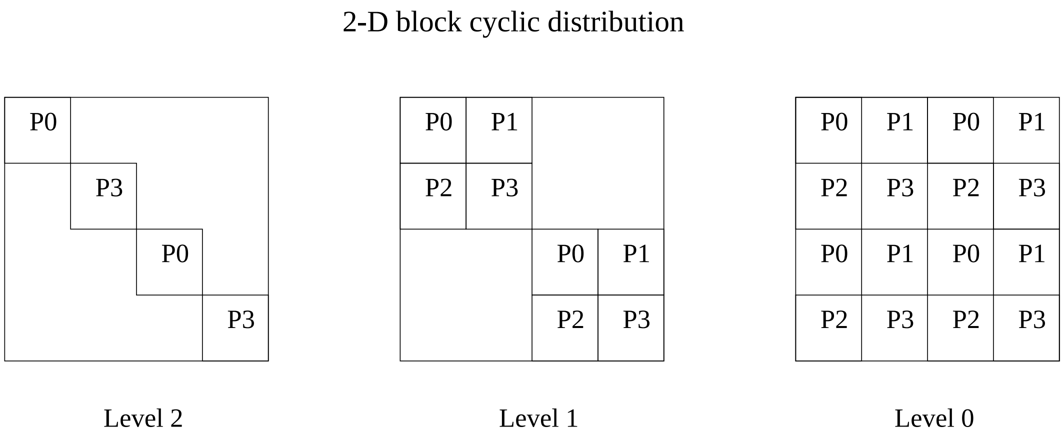

Figure 4.3: Active part of the matrix Q held by each process.

For a given rank-one update $Q(D + \rho z z^T)Q^T$ the processes that collaborate are those that hold a part of the global matrix Q. By contrast with previous implementations, with the two dimensional block cyclic distribution all the processes collaborate before reaching the top of the tree. We illustrate this in Figures 4.2 and 4.3 where the eigenvector matrix is distributed over 4 processes, using firstly a one-dimensional block distribution and secondly a two-dimensional block cyclic distribution. Suppose that, at level 1, all the deflations occur in the first submatrix, which is held by processes P_0 and P_1 for a one-dimensional block distribution and processes P_0, P_1, P_2, P_3 for a two-dimensional block distribution. With a one dimensional block distribution, processors P_0 and P_1 will have little computation to perform and then will have to wait for processes P_2 and P_3 before beginning the last rank-one update (level 0). With the two-dimensional block cyclic distribution, the computation is distributed among the 4 processes. This solves some of the load balancing problems that may appear when deflations are not evenly distributed among the processes.

Moreover, the two-dimensional block cyclic distribution is particularly well adapted for efficient and scalable parallel matrix-matrix multiplications. These operations are the main computational cost of this algorithm. In our parallel implementation, we use `PxGEMM`, as included in ScaLAPACK.

4.2 Orthogonality

Let $\hat{\lambda}$ be an approximate root of the secular equation (2.6). When we approximate the eigenvector u by replacing λ in (2.7) 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 $z_j/(d_j - \lambda)$. As a consequence, the computed eigenvector is very far from the true one and the resulting eigenvector matrix is far from being orthogonal.

Sorensen and Tang [30] 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 do not help and performance suffers.

The Gu and Eisenstat [19] approach based on the Löwner Theorem 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. On the other hand, the Löwner approach may require more communication than the extra precision approach, depending on how the parallelization is done. The reason is that the Löwner approach uses a formula that requires information about all the eigenvalues, requiring a broadcast, whereas the extra precision approach is "embarrassingly" parallel, with each eigenvalue and eigenvector computed without communication. However the extra communication of the Löwner approach is trivial compared with the communication of eigenvectors elsewhere in the computation.

The Löwner approach [19], [26] considers that the computed eigenvalues are the exact eigenvalues of a new rank-one modification $D + \rho\tilde{z}\tilde{z}^T$. By definition we have that

$$\det(D + \rho\tilde{z}\tilde{z}^T - \lambda I) = \prod_{j=1}^{n}(\hat{\lambda}_j - \lambda) \tag{4.1}$$

and also

$$\det(D + \rho\tilde{z}\tilde{z}^T - \lambda I) = \left(1 + \rho \sum_{j=1}^{n} \frac{\tilde{z}_j^2}{d_j - \lambda}\right) \prod_{j=1}^{n} (d_j - \lambda). \tag{4.2}$$

Then, combining (4.1) and (4.2) and setting $\lambda = d_i$ leads to

$$\tilde{z}_i = \sqrt{(\hat{\lambda}_i - d_i) \prod_{\substack{j=1 \\ j \neq i}}^{i-1} \frac{\hat{\lambda}_j - d_i}{d_j - d_i}}, \quad i = 1, \ldots, n.$$

If all the quantities $\hat{\lambda}_j - d_i$ are computed to high relative accuracy then $\tilde{z}_i$ can be computed to high relative accuracy. Substituting the exact eigenvalues $\{\lambda_i\}_{i=1}^n$ and the computed $\tilde{z}$ into (2.7) gives

$$\hat{u}_i = \left(\frac{\tilde{z}_1}{d_1 - \hat{\lambda}_i}, \ldots, \frac{\tilde{z}_n}{d_n - \hat{\lambda}_i}\right) \Bigg/ \sqrt{\sum_{j=1}^{n} \frac{\tilde{z}_j^2}{(d_j - \hat{\lambda}_i)^2}}\,.$$

Evaluating this formula, we obtain approximations of high componentwise accuracy to the eigenvectors $\hat{u}_i$ of $D + \tilde{z}\tilde{z}^T$. Provided that the eigenvalues $\hat{\lambda}_i$ are sufficiently accurate, which is assured by the used of a suitable stopping criterion when solving the secular equation, it can be shown (see [19]) that we obtain a numerical eigendecomposition $T \approx \widehat{U}\widehat{\Lambda}\widehat{U}^T$, that is, a decomposition with a relative residual of order ε and with $\widehat{U}$ orthogonal to working precision.

There are several ways to parallelize this approach. Either we consider it as an $O(n^2)$ operations cost, which means we can justify redundant computation in order to avoid communications, or we consider that the size of the data to communicate is negligible compare with that sent for the back transformation and then we can justify communications for distributing the work among the processes. We chose the latter approach.

Let S denote the set of processes that cooperate for a given rank-one update and k be the number of roots to approximate. Then each process of S computes k/S roots labeled $\{\hat{\lambda}_i\}_{i=i_0}^{i_{k_s}}$, the corresponding quantities $\hat{\lambda}_i - d_j, \quad i_0 \leq i \leq i_{ks}, \quad j = 1, \ldots, k$ and a part of each component of $\tilde{z}$:

$$\bar{z}_j = \prod_{i=i_0}^{i_{k_s}} (\hat{\lambda}_i - d_j) \prod_{i=i_0, j \neq i}^{i_{k_s}} (d_i - d_j)^{-1}.$$

Results are broadcast over S and processes update their $\tilde{z}$:

$$\tilde{z}_j = \prod_S \bar{z}_j.$$

Each process of S then holds the necessary information to compute its local part of $\hat{U}$ and no more communication is needed.

To compute the approximate eigenvalues and the quantities $\hat{\lambda}_j - d_i$ stably and efficiently, we use the hybrid scheme for the rational interpolation of $f(x)$ as developed by Li [25]. The hybrid scheme keeps the peak number of iterations relatively small for solving the secular equation. For our parallel implementation, this is helpful because the execution time for this part is determined by whichever eigenvalue takes the largest number of iterations.

4.3 Back Transformation

The main cost in the divide and conquer algorithm is in computing the product QU (see (2.4)). The efficiency of the whole implementation relies on a proper implementation of this back transformation. The goal is to reduce the size of the matrix-matrix multiplication when transforming the eigenvectors of the perturbed diagonal matrix to the eigenvectors of the tridiagonal matrix.

In this section, we explain a permutation strategy originally suggested by Gu [18] and used in the serial LAPACK divide and conquer code. Then we derive a permutation strategy more suitable for our parallel implementation. This new strategy is one of the major contributions of our work.

After the deflation process, we denote by G the product of all the Givens rotations used to set to zero component of z corresponding to nearly equal diagonal elements of D and by P the accumulation of permutations used to translate the zero components of z to the bottom of z:

$$PG(D + \rho z z^t)G^T P^T = \begin{pmatrix} \widetilde{D} + \rho \tilde{z}\tilde{z}^T & 0 \\ 0 & \bar{\Lambda} \end{pmatrix}. \tag{4.3}$$

Let $(\widetilde{U}, \widetilde{\Lambda})$ be the spectral decomposition of $\widetilde{D} + \rho \tilde{z}\tilde{z}^T$. 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 $T = Q(D + \rho z z^T)Q$ is obtained from

$$\begin{aligned} T &= Q(PG)^T PG(D + \rho z z^T)(PG)^T PGQ^T \\ &= Q(PG)^T U \Lambda U^T PGQ^T \\ &= W \Lambda W^T \end{aligned}$$

with $W = Q(PG)^T U$.

When not properly implemented, the computation of W can be very expensive. To simplify the explanation, we illustrate it with a 4×4 example: we suppose that $d_1 = d_3$ and that G is the Givens rotation used to set to zero the third component of z. The matrix P is a permutation that moves z_3 to the bottom of z. We indicate by a " $*$" that a value has changed.

There are two ways of applying the transformation $(PG)^T$, either on the left, that is on Q, or on the right, that is on U. Note that $Q = \mathrm{diag}(Q_1, Q_2)$ is block diagonal (see(2.3)) and so we would like to take advantage of this structure. It would halve the cost of the matrix multiplication if Q_1,Q_2 are of the same size. To preserve the block diagonal form of Q, we need to apply $(PG)^T$ on the right:

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

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. Consider again the 4×4 example:

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

At this step, a permutation is used to group the columns of 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 $\bar{U}(1{:}\,3,1{:}\,3)$. 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. In our parallel implementation, $\bar{P}$ groups the column of Q according to their local sparsity structure, that is, $\bar{P}$ permutes columns of Q belonging to the same process column. More precisely, locally, $\bar{P}$ puts together columns of Q with zero components in the lower part, then columns of Q without zero components, columns of Q with zero components in the upper part and finally columns of Q that are already eigenvectors. The resultant matrix $\bar{Q}$ has the following global structure:

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

$\bar{Q}_{11}$ 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,
$(\bar{Q}_{13}, \bar{Q}_{23})^T$ contains $k^{'}$ columns of Q_2 that correspond to deflated eigenvalues (they are eigenvectors of T),
$(\bar{Q}_{12}, \bar{Q}_{21})^T$ contains the $n - (n_1 + n_2 + k^{'})$ remaining columns of Q.

The matrix $\bar{U}$ has the structure

$$\bar{U} = \begin{array}{c c} & \begin{array}{cc} n{-}k^{'} & k^{'} \end{array} \\ \begin{array}{c} n{-}k^{'} \\ k^{'} \end{array} & \begin{pmatrix} \bar{U}_1 & 0 \\ 0 & I \end{pmatrix} \end{array}.$$

Then, for the computation of the product $\bar{Q}\bar{U}$, we use two calls to the parallel BLAS `PxGEMM` involving parts of U_1 and the matrices $(\bar{Q}_{11}, \bar{Q}_{12}), (\bar{Q}_{21}, \bar{Q}_{22})$. Unlike in the serial implementation, we can not assume that $k^{'} = k$, that is, that $(\bar{Q}_{13}, \bar{Q}_{23})^T$ 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. Let

$k(q)$ be the number of deflated eigenvalues held by the process column q, $0 \leq q \leq Q-1$. Then, $k^{'} = \min_{0 \leq q \leq Q-1} k(q)$. So, even if we can not perform matrix multiplies of minimal sizes as in the serial case, we still get good speed-up on many matrices.

5 The Divide and Conquer Code

The code is composed of 7 parallel routines `PxSTEDC`, `PxLAED0`, `PxLAED1`, `PxLAEDZ`, `PxLAED2`, `PxLAED3`, `PxLASRT` and it uses LAPACK's serial routines whenever possible.

`PxSTEDC` scales the tridiagonal matrix, calls `PxLAED0` to solve the tridiagonal eigenvalue problem, scales back when finished and sorts the eigenvalues and corresponding eigenvectors in ascending order by calling `PxLASRT`.

`PxLAED0` is the driver of the divide and conquer algorithm. It splits the tridiagonal matrix T into `TSUBPBS = (N-1)/NB +1` submatrices using rank-one modification. `NB` is the size of the block used for the two dimensional block cyclic distribution. It calls the serial divide and conquer code `xSTEDC` to solve each eigenvalue problem at the leaves of the tree. Then, each rank-one modification is merged by a call to `PxLAED1`:

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

`PxLAED1` is the routine that combines eigensystems of adjacent submatrices into an eigensystem for the corresponding larger matrix. It calls `PxLAEDZ` to form z as in (2.2), then calls `PxLAED2` to deflate eigenvalues and to group columns of Q following their sparsity structure as described in Section 4.3. Then, it calls `PxLAED3`, which distributes the work among the processes in order to compute the roots of the secular equation, solve the Löwner inverse eigenvalue problem and compute the eigenvectors of the rank-one update $\widetilde{D} + \rho \tilde{z} \tilde{z}^T$ (see (4.3)). Each root of the secular equation is computed by the serial LAPACK routine `xLAED4`. Finally, the eigenvector matrix of the rank-one update is multiplied into the larger matrix that holds the collective results of all the previous eigenvector calculations via the use of two calls to the parallel matrix multiplication routine`PxGEMM`.

6 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`).

`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`.

We use two types of test matrices. The first are symmetric matrices with random entries from a uniform distribution on $[-1, 1]$. The second type are generated by the ScaLAPACK subroutine `PxLATMS`. The matrix $A = U^T DU$, where U is orthogonal and $D = \mathrm{diag}(s_i, t_i)$ with $t_i \geq 0$ and $s_i = \pm 1$ chosen randomly with equal probability. For matrix 1, D has equally spaced entries from ε to 1. For matrix 2, D has geometrically spaced entries from ε to 1,

$$d_i = \pm \varepsilon^{\frac{i-1}{n-1}}, \qquad i = 1 : n,$$

and for matrix 3, D has clustered entries

$$d_i = \pm \varepsilon, \quad i = 1 : n-1, \qquad d_n = 1.$$

The type 3 matrices are designed to illustrate how B/II can fail to compute orthogonal eigenvectors.

Let $\widehat{Q}\widehat{\Lambda}\widehat{Q}^T$ be the computed spectral decomposition of A. To determine the accuracy of our results, we measure the scaled residual error and the scaled departure from orthogonality, defined by

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

[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.

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

Table 6.1: Normalized residual, normalized eigenvector orthogonality, and timing for a matrix of size $n = 1500$ on an IBM-SP2 (2x4 processor grid) for bisection/inverse iteration without and with reorthogonization, QR algorithm, and divide and conquer algorithm.

When both quantities are small, the computed spectral decomposition is the exact spectral decomposition of a slight perturbation of the original problem.

The tests were run on an IBM SP2 in double precision arithmetic. On this machine, $\varepsilon = 2^{-53} \approx 1.1 \times 10^{-16}$.

Table 6.1 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.

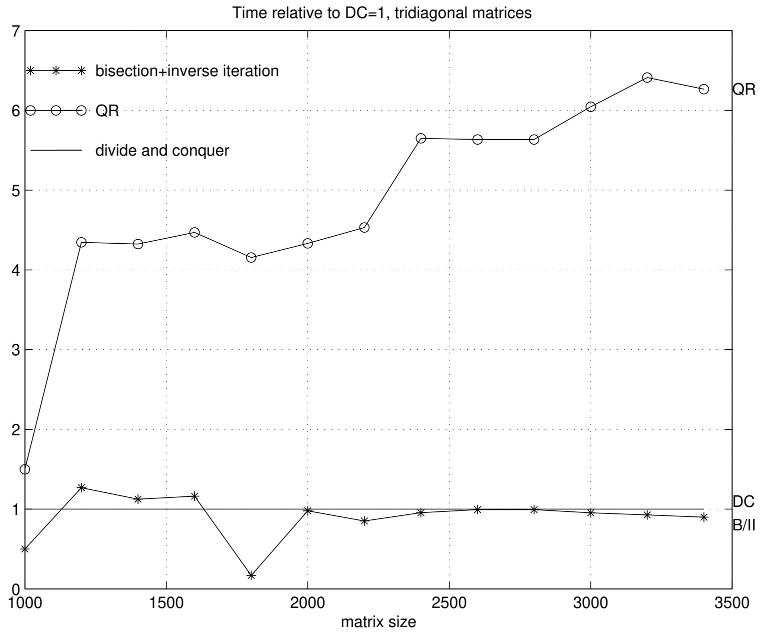

Figure 6.1: Execution times of PDTEBZ+PDSTEIN (B/II) and PDSTEQR2 (QR) relative to PDSTEDC (DC), on an IBM SP2, using 8 nodes. Tridiagonal matrices, eigenvalues of equally spaced modulus.

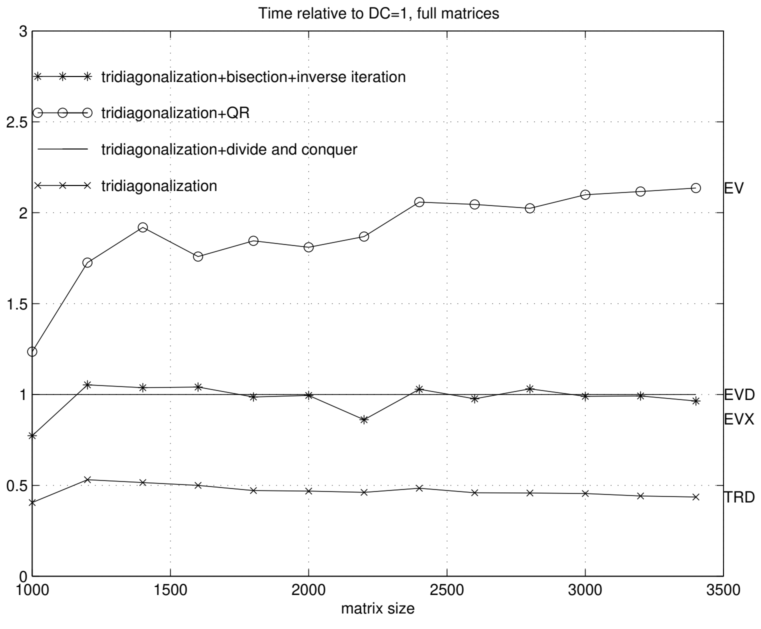

Figure 6.2: Execution times of PDSYEVX (B/II), PDSYEV (QR) and PDSYTRD (tridiagonalization) relative to PDSYEVD (DC), on an IBM SP2, using 8 nodes. Full matrices, eigenvalues of equally spaced modulus.

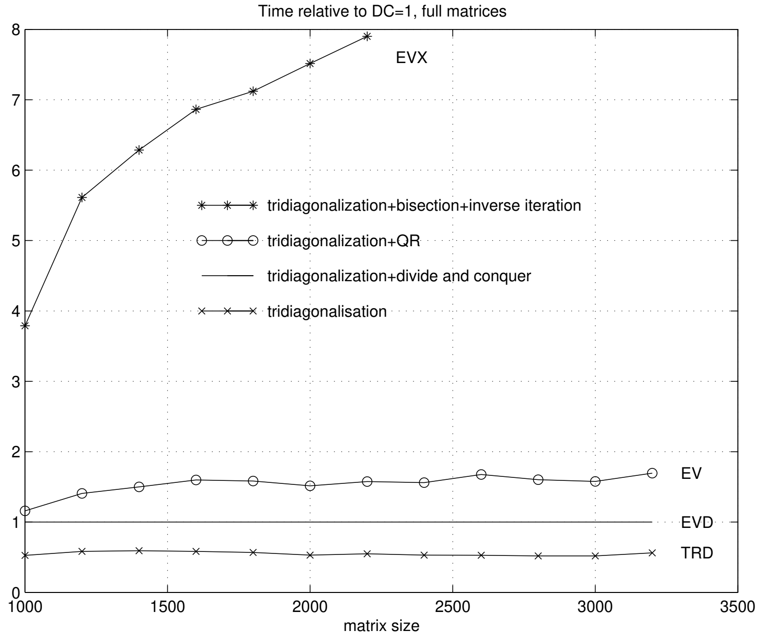

Figure 6.3: Execution times of `PDSYEVX` (B/II), `PDSYEV` (QR) and `PDSYTRD` (tridiagonalization) relative to `PDSYEVD` (DC), on an IBM SP2, using 8 nodes. Full matrices, eigenvalues of geometrically spaced modulus.

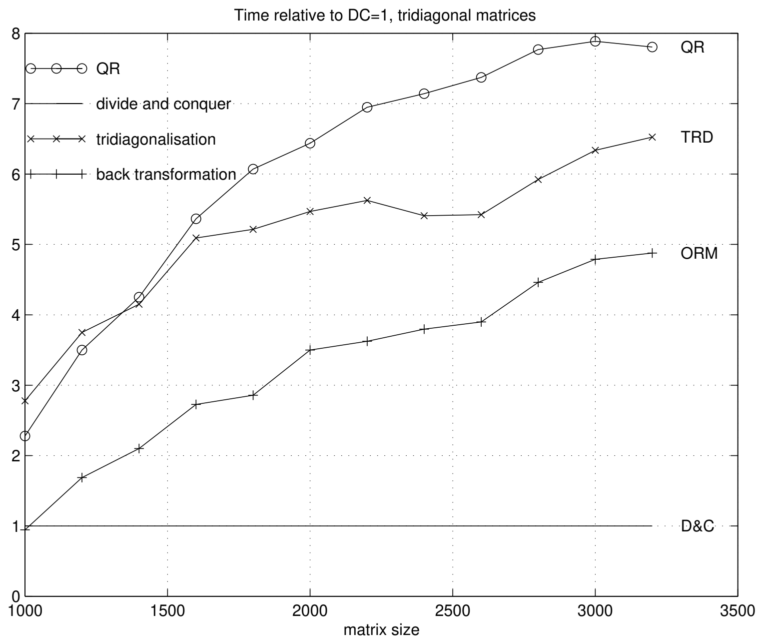

Figure 6.4: Execution times of `PDSTEQR2` (QR), `PDSYTRD` (tridiagonalization) and `PDORMTR` (back transformation) relative to `PDSTEDC` (DC). Measured on an IBM SP2, using 8 nodes. Tridiagonal matrices, eigenvalues of geometrically spaced modulus.

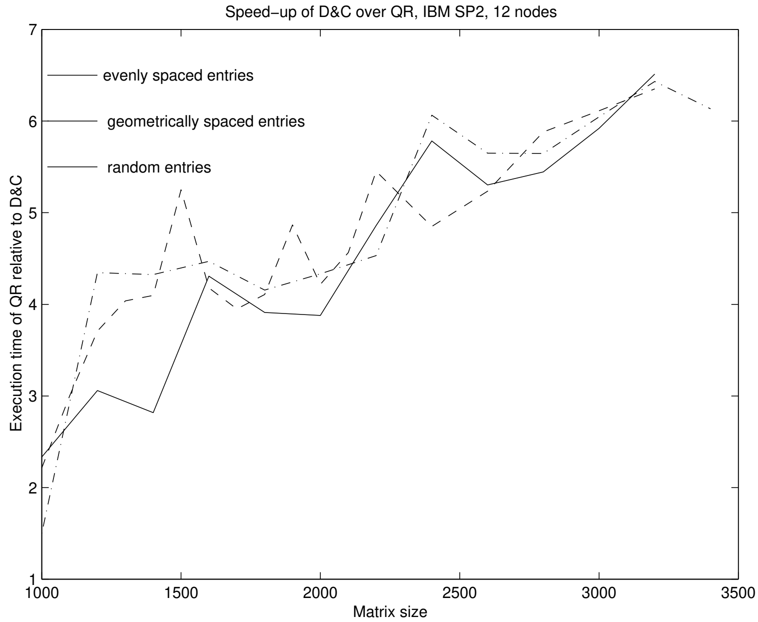

Figure 6.5: Speedups of PDSYEVD(DC) over PDSYEV(QR) with several types of full matrices. Tests done on an IBM SP2 using 12 nodes.

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 constant at 1. It is clear from Figures 6.1 and 6.2, 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 matrices in question are well separated. For inverse iteration, this situation 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 execution time between QR or B/II or divide and conquer is more noticeable in Figure 6.1 (up to a factor of 6.5) than in Figure 6.2 (up to a factor of 2.1) because the latter problem includes the overhead due to the tridiagonalization and back transformation processes.

As illustrated in Figure 6.3, divide and conquer runs faster than B/II as soon as eigenvalues are poorly separated or in clusters.

We also compare execution times of the tridiagonalization, QR and back transformation relative to the execution time of divide and conquer. ¿From Figure 6.4, it appears that when using the QR algorithm for the 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

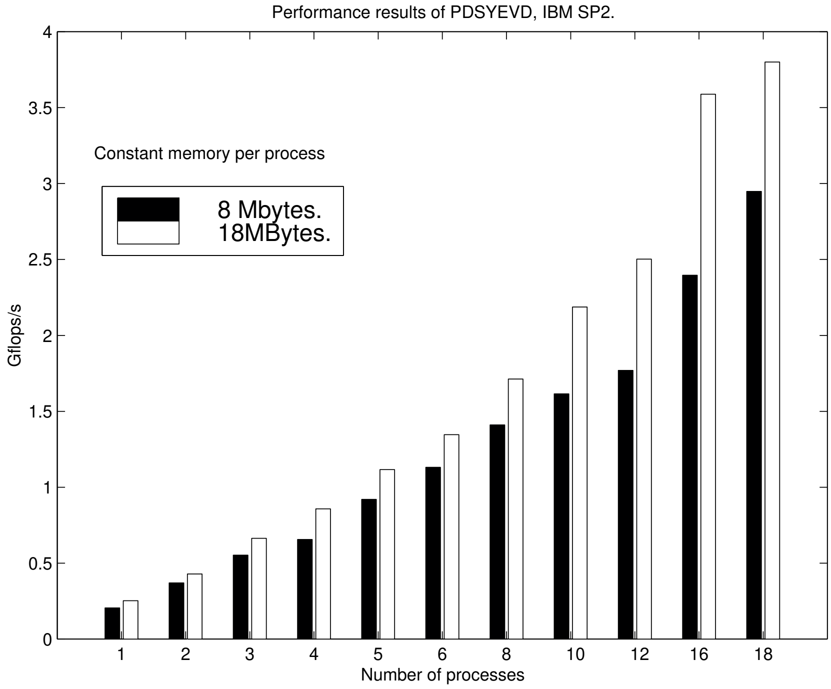

Figure 6.6: Performance of PDSTEDC, IBM SP2.

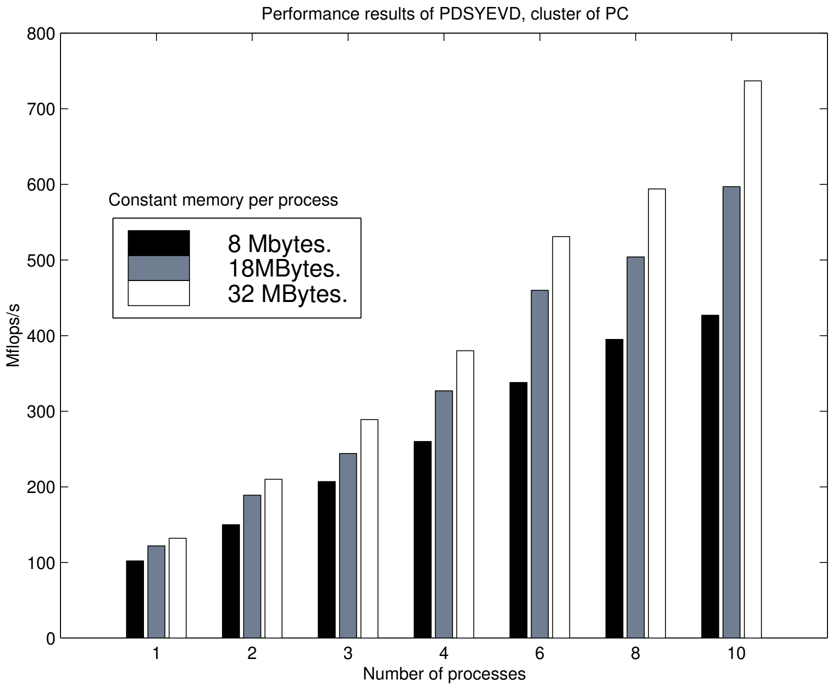

Figure 6.7: Performance of PDSEDC, cluster of 300 MHz Intel PII processors using a 100 Mbit Switch Ethernet connection.

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 [22] should be made to improve the tridiagonalization and back transformation.

We measured the performances of `PDSTEDC` on an IBM SP2 (Figure 6.6) and on a cluster of 300 MHz Intel PII processors using a 100 Mbit Switch Ethernet connection (Figure 6.7). 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) [32], which run at a peak of 440 Mflop/s on the SP2 and 190 Mflop/s on the PII. When using 18 processors on the SP2, our code runs at 50% of the peak performance of matrix multiplication and 40% on the cluster of PII's when using 10 processors. 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.

7 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 times of QR and bisection followed by inverse iteration available in the ScaLAPACK library.

Performance results on the IBM SP2 and a cluster of PC PII demonstrate the scalability and portability of our algorithm. Good efficiency is obtained mainly 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 [12] 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 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 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 developed in [31] 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], [13]. 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. H. Bischof, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling, A. McKenney, S. Ostrouchov, and D. C. Sorensen. *LAPACK Users' Guide, Release 2.0.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 1995.

[2] Zhaojun Bai, James W. Demmel, Jack J. Dongarra, Antoine Petitet, Howard Robinson, and K. Stanley. The spectral decomposition of nonsymmetric matrices on distributed memory parallel computers. *SIAM J. Sci. Comput.*, 18(5):1446–1461, 1997.

[3] Christian Bischof, Steven Huss-Lederman, Xiaobai Sun, Anna Tsao, and Thomas Turnbull. Parallel performance of a symmetric eigensolver based on the invariant subspace decomposition approach. In *proceedings of Scalable High Performance Computing Conference'94, Knoxville, Tennessee*, pages 32–39, May 1994. (Also 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] James R. Bunch, Christopher P. Nielsen, and Danny C. Sorensen. Rank-one modification of the symmetric eigenproblem. *Numer. Math.*, 31:31–48, 1978.

[6] Soumen Chakrabarti, James Demmel, and Katherine Yelick. Modeling the benefits of mixed data and task parallelism. Technical Report CS-95-289, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, May 1995. LAPACK Working Note 97.

[7] Jaeyoung Choi, Jack J. Dongarra, Roldan Pozo, and David 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, USA, November 1992. LAPACK Working Note 55.

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

[9] Kirsty F. F. Darbyshire. A parallel algorithm for the symmetric tridiagonal eigenproblem. Master's thesis, University of Manchester, 1994.

[10] James W. Demmel. *Applied Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997.

[11] James 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, USA, September 1994. LAPACK Working Note 86.

[12] Inderjit Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem.* PhD thesis, University of California, Berkeley, USA, 1997.

[13] Stéphane Domas and Françoise Tisseur. Parallel implementation of a symmetric eigensolver based on the Yau and Lu method. In *Vector and Parallel Processing – VECPAR'96*, Lecture Notes in Computer Science 1215, pages 140–153. Springer-Verlag, Berlin, 1997.

[14] J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. Stat. Comput.*, 8:139–154, 1987.

[15] Maria Paula Gonçalves Fachin. *The Divide-and-conquer Method for the Solution of the Symmetric Tridiagonal Eigenproblem and Transputer Implementations.* PhD thesis, University of Kent at Canterbury, 1994.

[16] K. Gates and P. Arbenz. Parallel divide and conquer algorithms for the symmetric tridiagonal eigenproblem. Technical report, Institute for Scientific Computing, ETH Zurich, 1994.

[17] Gene H. Golub. Some modified matrix eigenvalue problems. *SIAM Review*, 15(2):318–334, 1973.

[18] Ming Gu. *Studies in Numerical Linear Algebra.* PhD thesis, Yale University, 1993.

[19] Ming Gu and Stanley C. Eisenstat. A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem. *SIAM J. Matrix Anal. Appl.*, 15(4):1266–1276, 1994.

[20] Ming Gu and Stanley C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Matrix Anal. Appl.*, 16(1):172–191, 1995.

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

[22] Bruce Hendrickson, Elizabeth Jessup, and Christopher Smith. A parallel eigensolver for dense symmetric matrices. Submitted to *SIAM J. Sci. Comput.*, 1996.

[23] I. C. F. Ipsen and E. R. Jessup. Solving the symmetric tridiagonal eigenvalue problem on the hypercube. *SIAM J. Sci. Stat. Comput.*, 11(2):203–29, 1990.

[24] 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(2):530–548, 1994.

[25] Ren-Cang Li. Solving the secular equation stably and efficiently. Technical report, Department of Mathematics, University of California, Berkeley, CA, USA, April 1993. LAPACK Working Note 89.

[26] K. Löwner. Uber monotone matrix funktionen. *Math. Z.*, 38:177–216, 1934.

[27] A. Melman. A numerical comparison of methods for solving secular equations. *J. Comp. Appl. Math.*, 86(1):237–249, 1997.

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

[29] 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, USA, March 1994. LAPACK Working Note 69.

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

[31] Françoise Tisseur. Workspace reduction for the divide and conquer algorithm. Manuscript, November 1997.

[32] R. Clint Whaley and Jack Dongarra. Automatically tuned linear algebra software. Technical Report CS-97-366, Department of Computer Science, University of Tennessee, Knoxville, TN, USA, 1997. LAPACK Working Note 131.