Parallel divide and conquer algorithms for the symmetric tridiagonal eigenproblem

Author(s):
Gates, Kevin; Arbenz, Peter

Publication Date:
1994

Permanent Link:
https://doi.org/10.3929/ethz-a-001382257 →

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Wissenschaftliches Rechnen

Kevin Gates
Peter Arbenz

Parallel Divide and Conquer Algorithms for the Symmetric Tridiagonal Eigenproblem

October 1994

222

ETH Zürich
Departement Informatik
Institut für Wissenschaftliches Rechnen
Prof. Dr. W. Gander

Kevin Gates
Departement of Mathematics
University of Queensland
Brisbane, Queensland, 4072
e-mail: keg@maths.uq.oz.au

Peter Arbenz
Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich
e-mail: arbenz@inf.ethz.ch

This report is also available via anonymous ftp from ftp.inf.ethz.ch
as doc/tech-reports/1994/222.ps.

PARALLEL DIVIDE AND CONQUER ALGORITHMS FOR THE SYMMETRIC TRIDIAGONAL EIGENPROBLEM

KEVIN GATES[*] AND PETER ARBENZ[†]

Abstract. In this paper a new implementation of a divide and conquer algorithm will be considered. This algorithm, in contrast to the LAPACK algorithm, uses a different formulation of the update problem, and extended precision in order to maintain accuracy and orthogonality. Our Intel Paragon implementation shows, in contrast to the Hypercube implementation by Ipsen and Jessup [14], that good speedups can be obtained from a distributed memory parallel version of the divide and conquer eigenvalue algorithm .

Key words. symmetric tridiagonal eigenproblem, divide and conquer algorithm, parallel computing

AMS(MOS) subject classifications. 65F15, 65W05.

1. Introduction. Since the publication of Cuppen [8] divide and conquer algorithms have been investigated as a alternative to the traditional tridiagonal QR algorithm. Until recently, these implementations have had mixed success. The implementation of Dongarra and Sorensen [11] showed that one of these algorithm can be considerably faster than the traditional algorithm, even on single processor systems. It also showed the potential of these algorithms for parallel processing, in particular shared memory systems, where queues were used to distribute the work to all available processors. In contrast, an early implementation of this algorithm on a hypercube [14] indicated that the potential for these algorithms on shared memory systems was limited, due to an inability to balance the load across the available processors.

The second drawback of the early implementations of the divide and conquer algorithms was a stability problem inherent in the calculation of the eigenvectors. This problem resulted in computed eigenvectors which were not orthogonal. Since divide and conquer algorithms are recursive, and each stage depends on an accurate and orthogonal spectral decomposition from the previous stage, this failure to produce orthogonal eigenvectors could result in a complete failure of the algorithm. Only recently, two solutions for this problem were proposed and to some extent implemented. The first is the use of extended precision [17] and the second involves the use of an inverse eigenvalue problem to calculate some of the initial values and therefore accurate and orthogonal eigenvectors for a slightly perturbed problem [13],[12]. A code which is based on the second solution was recently included in LAPACK-2.0. This code is very fast, stable, and accurate, and should be considered the code of choice for sequential solutions of symmetric eigenvalue problems [16].

In this paper a new implementation of a divide and conquer algorithm will be considered. This algorithm, in contrast to the LAPACK algorithm, uses a different formulation of the update problem, and extended precision in order to maintain accuracy and orthogonality. In our implementation, the operations executed in extended

[*] Swiss Federal Institute of Technology (ETH), Institute for Scientific Computing, CH-8092 Zurich, Switzerland. Present adress: University of Queensland, Department of Mathematics, Brisbane, Queensland, 4072, Australia (`keg@maths.uq.oz.au`). The work of this author was supported by the Swiss National Science Foundation grant 5003-034435

[†] Swiss Federal Institute of Technology (ETH), Institute for Scientific Computing, CH-8092 Zurich, Switzerland (`arbenz@inf.ethz.ch`).

precision are implemented similarly to that proposed by Sorensen and Tang [17] but require considerably less storage space. Our implementation shows, in contrast to the implementation by Ipsen and Jessup [14], that good speedups can be obtained from a distributed memory parallel version of the divide and conquer eigenvalue algorithm.

2. The algorithm. Given a symmetric irreducible tridiagonal matrix T, compute the spectral decomposition

$$T = \begin{pmatrix} \tau_1 & \delta_2 & & \\ \delta_2 & \tau_2 & \ddots & \\ & \ddots & \ddots & \delta_n \\ & & \delta_n & \tau_n \end{pmatrix} = Y\Lambda Y^*, \qquad \Lambda = \operatorname{diag}(\lambda_1, \ldots, \lambda_n), \tag{2.1}$$

where $\mathbf{y}_i$, the i-th column of $Y = [\mathbf{y}_1, \ldots, \mathbf{y}_n]$, is the normalized eigenvector of T corresponding to the eigenvalue λ_i, i.e. $T\mathbf{y}_i = \lambda_i \mathbf{y}_i$.

There are three different ways to formulate a divide and conquer algorithm for the tridiagonal (or more generally banded) eigenvalue problem, rank-one restriction, rank-one modification and rank-one extension. The implementation of Dongarra and Sorensen [11] and the LAPACK implementation use the rank-one modification formulation originally proposed by Cuppen [8]. The extension formulation has the advantage of being easily adaptable to the generalized symmetric tridiagonal eigenvalue problem $T\mathbf{x} = \lambda S\mathbf{x}$ and was chosen for this implementation. Therefore, only the rank-one extension algorithm will be discussed, descriptions of the other algorithms can be found in [2].

The divide and conquer algorithms solve for each of the eigenvalue and corresponding eigenvector pairs independently. The final spectral decomposition of the original matrix is the combination of the independent results into a matrix of eigenvalues Λ and a matrix of corresponding eigenvectors Y, thus finding the spectral decomposition (2.1) is equivalent to solving $T\mathbf{y}_i = \lambda_i \mathbf{y}_i$ for $i = 1, \ldots, n$.

The first step of the rank-one extension algorithm is to permute the k-th row and column of T to the end of the matrix, where $k \approx \frac{n}{2}$. This permutation results in the following form for $T\mathbf{x} = \lambda\mathbf{x}$,

$$\begin{pmatrix} T_1 & O & \delta_k \mathbf{e}_k \\ O & T_2 & \delta_{k+1}\mathbf{e}_1 \\ \delta_k \mathbf{e}_k^* & \delta_{k+1}\mathbf{e}_1^* & \tau_k \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \xi_k \end{pmatrix} = \lambda \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \xi_k \end{pmatrix}, \qquad \begin{aligned} \mathbf{x}_1 &= (\xi_1, \cdots, \xi_{k-1})^*, \\ \mathbf{x}_2 &= (\xi_{k+1}, \cdots, \xi_n)^*, \end{aligned} \tag{2.2}$$

where T_1 and T_2 are the leading $(k-1 \times k-1)$ and the trailing $(n-k \times n-k)$ principal sub-matrices of T respectively. The spectral decompositions $Y_i^* T_i Y_i = \Lambda_i$, $i = 1, 2$, can be computed independently, and therefore simultaneously in parallel. Substituting these results into (2.2) and premultiplying by $Y_1 \oplus Y_2 \oplus 1$ gives

$$\begin{pmatrix} \Lambda_0 & \mathbf{b} \\ \mathbf{b}^* & \gamma \end{pmatrix} \mathbf{q} = \lambda \mathbf{q}, \tag{2.3}$$

where

$$\Lambda_0 := \Lambda_1 \oplus \Lambda_2, \quad \mathbf{b} = \begin{pmatrix} \delta_k Y_1^* \mathbf{e}_k \\ \delta_{k+1} Y_2^* \mathbf{e}_1 \end{pmatrix}, \quad \gamma = \tau_k, \quad \mathbf{q} = \begin{pmatrix} Y_1^* \mathbf{x}_1 \\ Y_2^* \mathbf{x}_2 \\ \xi_k \end{pmatrix}.$$

(2.3) defines the rank-one extension problem of interest. Observe that (2.3) defines an arrow matrix and that by choosing the appropriate sign for the columns of Y_1 and Y_2 all of the elements of $\mathbf{b}$ will be greater than or equal to zero.

Under certain conditions one or more diagonal elements of Λ_0 are also eigenvalues of (2.3). The process of identifying these eigenvalues and calculating the corresponding eigenvectors is called *deflation.* In order to precisely describe when we deflate, we define a tolerance $\varepsilon\tau$ where ε is the machine epsilon and τ is a given positive integer. The first, and simplest type of deflation arises when $\beta_j \leq \tau\varepsilon$ where β_j is the j-th element of $\mathbf{b}$. In this case the eigenpair $(d_j, \mathbf{q}_j)$ can be accepted as a fully accurate eigenpair of (2.3), and β_j is set to zero. The second type of deflation results from a similarity transformation with a Givens rotation G applied to (2.3) in order to make $\beta_j = 0$ for some j. In particular, let

$$[\mathbf{e}_j, \mathbf{e}_i]^* G [\mathbf{e}_j, \mathbf{e}_i] = \begin{pmatrix} \gamma & -\sigma \\ \sigma & \gamma \end{pmatrix}, \qquad \sigma = \frac{\beta_j}{\sqrt{\beta_i^2 + \beta_j^2}}, \quad \gamma = \frac{\beta_i}{\sqrt{\beta_i^2 + \beta_j^2}}. \tag{2.4}$$

Then, (2.3) is replaced by

$$G \begin{pmatrix} \Lambda_0 & \mathbf{b} \\ \mathbf{b}^* & \gamma \end{pmatrix} G^* G \mathbf{q} = \lambda G \mathbf{q}$$

provided that the off-diagonal element that is created in the the arrow matrix by the transformation is sufficiently small. In order to be a negligible perturbation to the original matrix this element must not be larger that ε in modulus. Thus, the deflation criterion is $|\sigma\gamma(\lambda_i - \lambda_j)| \leq \tau\varepsilon$. After deflation the resulting eigenpair of (2.3) is $(\gamma^2\lambda_j + \sigma^2\lambda_i, \sigma\mathbf{q}_i + \gamma\mathbf{q}_j)$.

After the deflation process we arrive – after some permutations – at an eigenvalue problem of the form

$$\begin{pmatrix} \Lambda & 0 \\ 0 & A \end{pmatrix} \mathbf{q} = \lambda \mathbf{q}, \qquad A = \begin{pmatrix} \Omega & \mathbf{b} \\ \mathbf{b}^* & \gamma \end{pmatrix}, \qquad \mathbf{q} = \begin{pmatrix} 0 \\ \mathbf{q} \end{pmatrix}, \qquad A \in \mathbb{R}^{m \times m} \tag{2.5}$$

where $\beta_k > 0$ for all k and as a consequence of deflation $\Omega = \operatorname{diag}(\alpha_1, \ldots, \alpha_{m-1})$, $\alpha_1 > \alpha_2 > \cdots > \alpha_{m-1}$. A is an *irreducible arrow* matrix. For the rest of this discussion it is assumed that the original matrix has been deflated and that the arrow matrix A is irreducible.

It is easy to verify that for $\lambda \notin \sigma(\Omega)$

$$S^* \begin{pmatrix} \Omega - \lambda & \mathbf{b} \\ \mathbf{b}^* & \gamma - \lambda \end{pmatrix} S = \begin{pmatrix} \Omega - \lambda & \mathbf{0} \\ \mathbf{0}^* & -f(\lambda) \end{pmatrix}, \qquad S = \begin{pmatrix} I_{m-1} & -(\Omega - \lambda)^{-1}\mathbf{b} \\ \mathbf{0}^* & 1 \end{pmatrix}, \tag{2.6}$$

where

$$f(\lambda) = \lambda - \gamma + \mathbf{b}^*(\Omega - \lambda)^{-1}\mathbf{b} = \lambda - \gamma + \sum_{j=1}^{m-1} \frac{\beta_j^2}{\alpha_j - \lambda}. \tag{2.7}$$

So, λ is an eigenvalue of A if and only if $f(\lambda) = 0$. As the derivative of f is bounded below,

$$f'(\lambda) = 1 + \sum_{j=1}^{m-1} \frac{\beta_j^2}{(\alpha_j - \lambda)^2} \geq 1,$$

the $m-1$ poles of f strictly interlace the m zeros λ_j of f,

$$\lambda_1 > \alpha_1 > \lambda_2 > \alpha_2 > \cdots \lambda_{m-1} > \alpha_{m-1} > \lambda_m.$$

From (2.6) one sees that a normalized eigenvector corresponding to the eigenvalue λ_k is given by

$$\mathbf{q}_{\lambda_k} = \frac{S\begin{pmatrix}\mathbf{0}\\1\end{pmatrix}}{\left\|S\begin{pmatrix}\mathbf{0}\\1\end{pmatrix}\right\|} = \begin{pmatrix}\beta_1/(\lambda_k-\alpha_1)\\ \vdots \\ \beta_{n-1}/(\lambda_k-\alpha_{m-1})\\ 1\end{pmatrix}\Bigg/\sqrt{f'(\lambda_k)}\,. \tag{2.8}$$

We note that $\mathbf{q}_{\lambda_k}$ is an intermediate result from of the calculation of $f(\lambda_k)$.

Numerous zero finders have been proposed for solving the *secular equation* $f(\lambda) = 0$ [7],[11],[6]. They all are based on a rational approximate of f, and are either quadratically or cubically convergent. All of these zero finders obtain zeros which satisfy (2.9) [17],[12],[6];

$$|\text{computed}(\lambda_k) - \lambda_k| < \kappa\varepsilon m\|A\|, \qquad \kappa = \mathcal{O}(1). \tag{2.9}$$

Notice that the calculation of any two zeros is independent, thus any or all zeros can be found independently and in parallel. This parallelism in the zero finder makes this algorithm very attractive for shared memory machines, where job queues enable an efficient implementation of this part of the algorithm.

The spectral decomposition of T is now found by multiplying the eigenvector matrices,

$$T = (Y_1 \oplus Y_2 \oplus 1)G^*(I\oplus Q)\Lambda(I\oplus Q)^*G(Y_1\oplus Y_2\oplus 1)^*, \tag{2.10}$$

where $\Lambda = \operatorname{diag}(\lambda_1 \ldots \lambda_n)$ and G is the product of all the Givens rotations in the deflation process. This matrix-matrix multiplication is a BLAS 3 operation, which makes the algorithm very efficient on the current generation of microprocessors.

The algorithm described is one step of this divide and conquer algorithm. In (2.3), it was assumed that the spectral decomposition of T_i for $i = 1, 2$ was known. In a pure implementation these decompositions are calculated with the same algorithm, which is continued until the subproblems are 2×2 or 1×1 blocks, and can be solved directly. A second variant of this algorithm is to continue splitting the problems until all of the subproblems are less than or equal to some $q \ll n$. The algorithm is completed by solving the subproblems with the tridiagonal QR algorithm. In the Dongarra-Sorensen implementation q was chosen to be $n/8$, which means that a three stage algorithm was used. Choosing $q = n/8$ does not limit the number of processors which can be used, since in a shared memory machine all the zero finding work can be done in parallel by all available processors.

3. Accuracy.

As was mentioned above the eigenvalues are found to sufficient accuracy; however, the use of (2.8) to calculate the eigenvectors can result in inaccurate results [3]. This situation arises when β_k, for some k is small, but not small enough that λ_k can be found through deflation. In order to illustrate this situation assume that the new $\lambda_k \approx \alpha_k$. The function

$$f_k(\lambda) := f(\lambda) - \frac{\beta_k^2}{\alpha_k - \lambda} = \lambda - \gamma + \sum_{j=1, j\neq k}^{m-1} \frac{\beta_j^2}{\alpha_j - \lambda}.$$

is bounded in a sufficiently small neighborhood of α_k. λ_k is now expressed as a perturbation expansion in β_k,

$$\begin{aligned} \lambda_k &= \alpha_k + \beta_k^2/f_k(\alpha_k) + \mathcal{O}(\beta_k^4), && \text{if } f_k(\alpha_k) > 0, \\ \lambda_k &= \alpha_k + \beta_k/\sqrt{f_k'(\alpha_k)} + \mathcal{O}(\beta_k^2), && \text{if } f_k(\alpha_k) = 0. \end{aligned} \tag{3.1}$$

Note that if $f_k(\alpha_k) < 0$, λ_{k+1} is the eigenvalue closest to α_k. If $f_k(\alpha_k) = 0$ there is a twin eigenvalue $\lambda_{k+1} \cong \alpha_k - \beta_k/\sqrt{f_k'(\alpha_k)}$. As $f_k(\alpha_k) \to 0$ by continuity there will exist an eigenvalue pair that is close to α_k.

According to (2.9), λ_k, and thus $\lambda_k - \alpha_k$, can be accurately determined to $\mathcal{O}(\varepsilon)$. However, (3.1) indicates that the actual difference $\lambda_k - \alpha_k$ can be much smaller when $\beta_k < \sqrt{\epsilon}$. In the case where $\beta_k \cong \varepsilon$ and $f_k(\alpha_k) \cong \varepsilon$ then $\lambda_k - \alpha_k \cong \varepsilon$ and only one or two digits of accuracy are obtained. The formula for the eigenvectors, (2.8), calculates the elements of $\mathbf{q}_{\lambda_k}$ using the relative difference of $\lambda_k - \alpha_j$. This means that for $\lambda_k \cong \lambda_{k+1}$ the corresponding eigenvectors will be guaranteed orthogonal only to one or two digits. In general, the error in $\mathbf{q}_{\lambda_k}$ is directly related to the *relative* error in $\lambda_k - \alpha_k$ and the eigenvectors computed with (2.8) and (2.10) may not be orthogonal.

Two ways have been proposed to avoid this loss of orthogonality. The first way is to compute λ_k to high relative accuracy such that

$$|\text{computed}(\lambda_k) - \alpha_j| < \kappa\varepsilon|\lambda_k| \quad \forall j = 1, \ldots, m \qquad \kappa = \mathcal{O}(1). \tag{3.2}$$

This will require evaluating (2.7) in double the working precision [17]. A second way to insure that the eigenvectors are accurate and orthogonal is to *recompute* the entries β_k and γ of the arrow matrix (2.3) by well-known formulae from inverse eigenvalue problems [12],[5],

$$\hat{\gamma} = \sum_{j=1}^{m} \hat{\lambda}_j - \sum_{j=1}^{m-1} \alpha_j, \qquad \hat{\beta}_k^2 = \prod_{j=1}^{m} (\alpha_k - \hat{\lambda}_j) \Big/ \prod_{\substack{j=1 \\ j \neq k}}^{m-1} (\alpha_k - \alpha_j), \quad 1 \le k < m \tag{3.3}$$

where $\hat{\lambda}_j$ denotes the *computed* eigenvalues of A. Using the arrow matrix $\hat{A}$ constructed in the same way as A from the α_k, $\hat{\beta}_k$, and $\hat{\gamma}$ instead of A is justifiable since $||A - \hat{A}|| < \kappa\epsilon m^2 ||A||$. The modified equation (2.8) gives the eigenvectors of $\hat{A}$ to working precision. The obtained eigenvectors are orthogonal and good approximations to the desired eigenvectors of A.

4. Parallelism.

There are many different issues to be solved in order to parallelize this program. The first issue is the different types of potential parallelism which exist within the algorithm. Since the divide and conquer algorithms are recursive the natural structure of an implementation is a binary tree, which can be used to naturally split the work among the processors. At the root of the tree all processors should cooperate to solve the full problem, and at each branch of the tree the processors are naturally split in half. On the other hand the parallelism of the zero finding is natural for a task pool for a set number of processors. A load balance problem that may arise is when the deflation of the divide and conquer algorithm is not evenly distributed across the branches of the tree and therefore the available processors. This could result in considerable speedup for some processors and none for others. In this case the faster processors will have to wait for the other processors thus potentially destroying the speedup gained through the use of the tree.

A second issue is how matrices should be distributed for this algorithm. Since the algorithm is dominated by matrix-matrix multiplication it would seem natural to distribute the matrices in a block wrapped fashion (see [10] for a discussion of this distribution). However, the k-th vector of Q is naturally calculated by the processor which calculates λ_k. This could imply a redistribution of Q, a duplication of work involved in the calculation of $(\lambda_k, \mathbf{q}_k)$, or a different matrix distribution.

A third issue is which method to use in order to maintain accurate and orthogonal eigenvectors.

4.1. Work Distribution. The first problem which will be considered is the problem raised by [14] and [10]. These papers theorize that if the natural tree structure of the algorithm is used, and deflation is observed by one branch of the tree, then this will result in an unequal distribution of work in the last and dominant stage of the algorithm. The deflation effect combined with an unequal number of iterations by the zero finder will result in a greatly reduced parallel efficiency.

In order to more fully investigate this problem it is assumed that the tree structure is used. This means that at each stage of the parallel algorithm the number of available processors is divided by two, and that the one half of the processors work on the first subproblem, and the second half work on the second subproblem. This division of work continues until every processor is responsible for the completion of a subproblem. In addition, it is assumed that all of the processors working on a subproblem divide the work equally or that a good work sharing mechanism is implemented. The third aspect of this problem, unequal number of iterations of the zero finder, has been solved by improving the zero finder.[15]

In a divide and conquer algorithm for most of the work will be at the root of the tree, where it will be parallelized across the zero finding tasks. We will assume that this work is ideally parallelized. Now consider the worst case situation for one stage of this algorithmic implementation: one subproblem has considerable deflation and the second subproblem has no deflation. In this situation the time required to solve the second subproblem will dominate the solution time. If this unbalanced deflation occurs at each stage of the complete algorithm then the solution time will be determined by the path without deflation. To quantify this loss of efficiency let n = *problem size*, p = *number of processors*, α = *zero finder iterations* and $T(p)$ = *time in Flops for p processors*, then

$$\begin{aligned} T(p) &= \frac{n^3+\alpha n^2}{p} + \frac{(n/2)^3+\alpha(n/2)^2}{(p/2)} + \dots &< \frac{4n^3}{3p} + \frac{2\alpha n^2}{p} \\ T(1) &= n^3+\alpha n^2 + (n/2)^3 + \alpha(n/2)^2 + \dots &< \frac{8}{7}n^3 + \frac{4}{3}\alpha n^2. \end{aligned} \tag{4.1}$$

By using the common definition of efficiency (E),

$$E = \frac{T(1)}{pT(p)} \approx \frac{2(6n+7\alpha)}{7(2n+7\alpha)} \approx \frac{6}{7}, \tag{4.2}$$

it can be seen that a divide and conquer algorithm should still have about 85% efficiency with maximally unbalanced deflation. This is so because most of the work is concentrated in the root node of the binary tree. Notice that we assumed in our analysis that the work corresponding to one node in the tree is well balanced among the involved processors.

For typical matrices the efficiency will be higher since the deflation will not be so unbalanced.

4.2. Matrix Distribution. The issue in this section is which matrix distribution is appropriate for use with this algorithm. In [10] a block layout is recommended for matrix multiplication and linear algebra routines and efficient matrix-matrix multiplication algorithms are given. However, since the corresponding eigenvectors are computed as side products during the zero finding procedure, the matrix $(I \oplus Q)$ in (2.10) is naturally distributed by columns. If this distribution is not desired then the matrix must be redistributed or the zero finder work must be duplicated in the processors which should contain part of a column of Q.

A second choice for the matrix distribution is to use the natural distribution of $(I \oplus Q)$ and to distribute the other matrix by rows, so that the matrix-matrix multiplication can be done in place. This type of matrix distribution causes more communication in the multiplication routine, but is much simpler to implement.

There are two global dependencies in this divide and conquer algorithm. The first is that all processors which are cooperating to solve for the zeros (eigenvalues) and associated eigenvectors need to know $\mathbf{b}$, Ω and γ. The second global dependency is the matrix-matrix multiplication. These two global dependencies imply two synchronization barriers (communication points) in the parallel program.

Consider the block distribution of the matrices and associated vectors on a $\sqrt{p}\times\sqrt{p}$ grid of processors. $\mathbf{b}$, Ω and γ are needed on all processors. Ω can be communicated within the previous matrix-matrix multiplication, and is therefore communication free. $\mathbf{b}$ can not be calculated until the multiplication is finished, and is distributed across a row of processors. With $\sqrt{p}$ communications $\mathbf{b}$ can be communicated across the row, and with an additional $\sqrt{p}$ communications across the columns, all of the processors have the information required to find the new zeros. These communications can not be hidden by floating point operations because $\mathbf{b}$ is required for the next stage of the algorithm. The matrix-matrix multiplication can be accomplished with Cannon's algorithm from [10] which will take at most $4\sqrt{p}$ communications. These communications can be hidden with floating point operations if asynchronous communication is available.

TABLE 4.1
Communication costs for divide and conquer algorithm.

	Vector Comm.	Matrix multiplication Comm.	Matrix multiplication Floating Point
		Block Version	
Sync	$2\sqrt{p}\times n$	$4\sqrt{p}\times(n/p)^2$	$\sqrt{p}\times(n/p)^3$
Async	$2\sqrt{p}\times n$	—	$\sqrt{p}\times(n/p)^3$
		Row-Column Version	
Sync	2 Broadcast	$p\times n^2/p$	$p\times n^3/p^2$
Async	2 Broadcast	—	$p\times n^3/p^2$

For the second suggested distribution with p processors, $\mathbf{b}$, Ω and γ are calculated and available in two processors, so the communication of these values can be accomplished with broadcasts (which should be less expensive than normal communications) at a cost of at most p communications. The matrix-matrix multiplication will require $p-1$ communications. With this distribution asynchronous message passing can once again be used to hide the cost of the matrix multiplication. A summary of the resulting costs for both matrix distributions is included in Table 4.1.

For a parallel processor with asynchronous communications the advantage of the block distribution is limited, since most of the communications can be hidden by computations. For this reason and since the block distribution introduces considerable code complexity, the row-column distribution was used in this implementation.

4.3. Extended Arithmetic versus Gu and Eisenstat. The decisions made in the previous two sections, over the distribution of the matrix, and the necessity of distributing the work over the processor grid lead to the decision to use extended arithmetic. If the eigenvector matrix of A is kept in place, and not calculated every time it is needed, then the extended arithmetic will require two synchronization steps. The first to communicate needed values for the next stage of computations and the second when the matrices are multiplied. On the other hand, the Gu-Eisenstat algorithm would require three communications, an additional communication is required in order to communicate the new eigenvalues to all the processors working on a particular subproblems, in order to update $\mathbf{b}$ and γ in (3.3). This additional synchronization comes before the matrix multiplication and is an all to all communication, since all of the processors are involved in calculating the eigenvalues, and the eigenvectors. If work is duplicated in the zero finding calculations, the additional synchronization would be an all to all communication within the rows of the processor array. All to all communications are expensive, as is the additional synchronization, therefore extended arithmetic was used in this implementation to maintain accuracy and orthogonality of the eigenvectors.

5. Implementation Details.

In this section many of the implementation details of this particular divide and conquer algorithm will be discussed. These details are discussed because they can have a large effect on the eventual performance of the algorithm.

The first implementation detail to be discussed is deflation and the application of the resulting permutation and Givens rotations in the calculation of the eigenvector matrix. In this implementation a deflation tolerance of $\tau = 30 \max_i |\alpha_i|$ was used. This program does not produce relatively accurate eigenvalues in the sense of [4], instead the eigenvalues are absolutely accurate with respect to the largest eigenvalue. In this implementation the eigenvector matrix of the previous stage was never changed, that is all Givens rotations and perturbations were applied to the eigenvector matrix of the inner problem.

The second important implementation detail of this program is the extended precision evaluation of the secular equation and the termination criterion. In this program the termination criterion for the zero finder was a relative accuracy of $10\varepsilon|\alpha_k - \lambda|$, where α_k is the pole closest to λ. The working precision zero finder used was from [6], since it is simple and effective, although it can take more operations than the zero finder due to Li [15]. The decision to proceed to extended precision is made by the following termination criterion, which indicates when no additional accuracy can be gained with the zero finder in working precision:

$$f(\lambda_k) < \varepsilon \left(3|\lambda_k - \sigma| + (3n+17) \sum_{j=1}^{m} \frac{\beta_j^2}{|\alpha_j - \lambda_k|} \right). \tag{5.1}$$

The extended precision zero finder performs a Newton iteration with a bisection safeguard. The roots are already close to convergence, and the safeguard is inexpensive and prevents a jumping out of the appropriate interval, which can happen with the Newton method. Other more complex zero finders, including the one used in normal precision, were not implemented in extended precision due to the cost and unnecessary code complexity, particularly since in most cases only one step of the extended precision zero finder is necessary to obtain full accuracy.

The evaluation of the secular equation in extended precision is an important implementation issue for this program. In order to do so the program simulates an extended double precision (that is double the working precision). This simulation requires splitting a variable into two working precision variables, one that contains the upper half of the significant digits and the second that contains the lower half of the significant digits. This was accomplished with an idea due to Dekker [9] that is contained in Algorithm 5.1.

Algorithm 5.1.

```
$\zeta = base^{digits/2} + 1$
$\pi = a * \zeta;\ \xi = a - \pi$          Define temporary variables.
$a^{(u)} = \xi + \pi$                    Calculate upper half.
$a^{(l)} = a - a_1$                      Calculate lower half.
$return(a^{(u)}, a^{(l)})$               Return variables containing upper and lower halves.
```

Now the following algorithm for evaluating the secular equation in extended precision can be presented:

Algorithm 5.2.

```
For $i = 1 : m$
    $(b^{(u)}, b^{(l)}) := b_i;$                                   Split $\mathbf{b}_i$.
    $w_i := \delta_i - \xi;$                                        Calculate $\mathbf{w}_i\alpha_i - \lambda = \delta_i - \xi$
    $(w^{(u)}, w^{(l)}) := w_i;$                                   Split $w_i$.
    $w^{(e)} := (\delta_i - w_i) - \xi;$                              Calculate remainder of $\delta_i - \xi$
    if $(i < m)$ $v_i := b_i/w_i;$                                  Calculate $\mathbf{v}_i$
    else $v_i := w_i;$                                              Calculate $\mathbf{v}_n$
    $(v^{(u)}, v^{(l)}) := v_i;$                                   Split $\mathbf{v}_i$.
    $b^{(e)} := (((b_i - w^{(u)}v^{(u)}) - w^{(u)}v^{(l)}) - w^{(l)}v^{(u)}) - w^{(l)}v^{(l)};$
                                                                  Calculate remainder of $\mathbf{b}_i - \mathbf{v}_i\mathbf{w}_i$
    $\pi^{(u)} := b^{(u)}v^{(u)};\ \pi^{(l)} := b^{(l)}v^{(u)} + b^{(u)}v^{(l)};$    Calculate products $\pi^{()}$ from $b^{()}v^{()}$
    $\pi^{(e)} := b^{(l)}v^{(l)};$                                   Calculate product $\pi^{(e)}$
    $\tau^{(u)} := \pi^{(l)} + \pi^{(u)};$                            Calculate first term of product.
    $\tau^{(l)} := ((\pi^{(u)} - \tau^{(u)}) + \pi^{(l)}) + \pi^{(e)};$       Calculate second product term.
    if $(i < m)$ $\tau^{(l)} := \tau^{(l)} + v_i b^{(e)} - v_i^2 w^{(e)};$      Add second order corrections.
    else { $\tau^{(u)} := -\delta_i;\ \tau^{(l)} := \xi;$}                  Calculate terms for $i = n$
    if $(|\sigma^{(u)}| > |\tau^{(u)}|)$ { $\zeta^{(u)} := \sigma^{(u)};\ \zeta^{(l)} := \tau^{(u)};$ }
    else { $\zeta^{(u)} := \tau^{(u)};\ \zeta^{(l)} := \sigma^{(u)};$ }
    $\sigma^{(u)} := \zeta^{(u)} + \zeta^{(l)};$                         Calculate sum $\sigma^{(u)}$
    $\sigma^{(l)} := ((\zeta^{(u)} - \sigma^{(u)}) + \zeta^{(l)}) + \tau^{(l)};$     Calculate sum $\sigma^{(l)}$
return $(\sigma^{(u)} + \sigma^{(l)})$                                 Return the sum $(\sigma^{(u)} + \sigma^{(l)})$
```

Notice that both Algorithms 5.1 and 5.2 depend on IEEE consistent arithmetic.

As was discussed in Section 4.2, considerable communication costs were hidden through the use of asynchronous communications in the matrix-matrix multiplication routine. This was accomplished by using two message buffers and initiating the communications before any floating point operations were done. In this way the data required for the next step of the multiplication was received during the multiplication of the previous step. Since this could not be done with the communication of $\mathbf{b}$, Ω and γ, these items were communicated with an asynchronous broadcast.

The matrix-matrix multiplication in this implementation is accomplished by passing the calculated columns of the eigenvectors in a ring. Since the left hand matrix was stored by rows, a complete update of the matrix part stored in a processor could be calculated.

6. Results – Sequential and Parallel. The sequential version of this program was tested with the LAPACK matrix test suite. This suite includes a matrix with random elements, matrices with geometrically and arithmetically distributed eigenvalues, and a matrix with a large cluster of eigenvalues. Three different comparisons were made with each of the test matrices. The first two comparisons concerned the accuracy of the computation: the size of the residual matrix $AY - Y\Lambda$ and the orthogonality of the eigenvectors was checked. The third was a timing comparison. In mathematical terms the following tests were made

$$\text{Norm } = \max_j |A\mathbf{y}_j - \lambda_j\mathbf{y}_j|_2 \qquad \text{Orth } = \max_j |Y^*\mathbf{y}_j - \mathbf{e}_j|_2. \tag{6.1}$$

Figure 6.1 is a graph of these two values for an entire range of matrices for the random dense matrix. Observe that these results are in relationship to ε. These accuracy

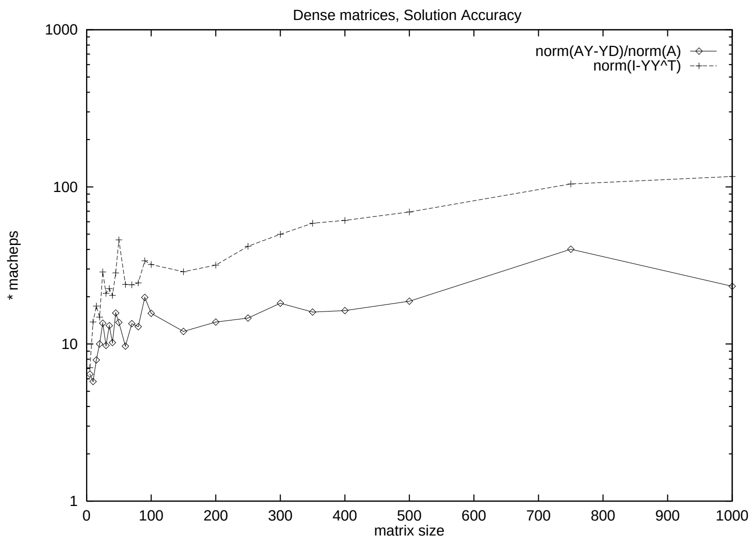

FIG. 6.1. *Solution Accuracy*

and orthogonal results are representative of the results from all of the test matrix suites, including the runs on the Intel Paragon parallel computer. For this reason the remainder of this section will only consider timing results.

In the timing results we will consider first the sequential times, and show that the times from the sequential version of this program are good enough to consider a parallel version of this program. In order to demonstrate the potential of this program comparisons have been made with the other solution methods for the symmetric eigenvalue problem. The results for two different matrices which are representative of the performance of the program have been printed in Figures 6.2 and 6.3.

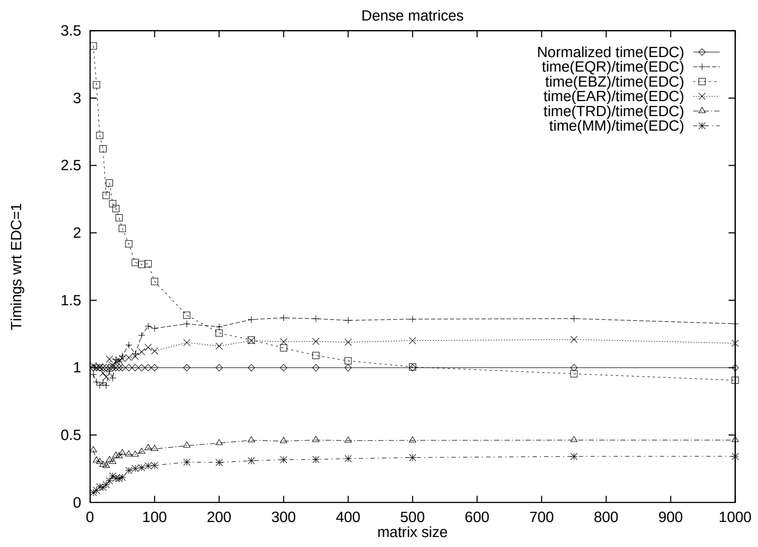

FIG. 6.2. *Timing Results for the Random Matrix*

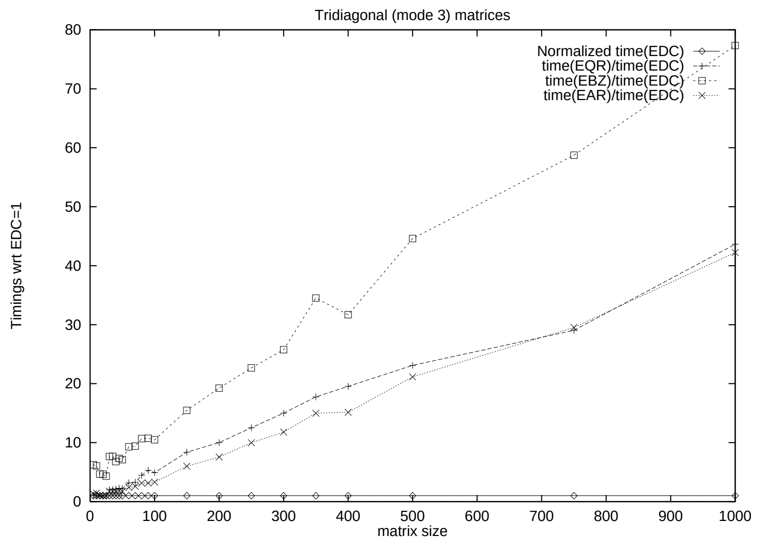

FIG. 6.3. *Timing Results Tridiagonal with Algebraically Distributed Eigenvalues*

In Fig. 6.2 all results are normalized by the fastest sequential routine, **EDC**, the divide and conquer routine from the University of California at Berkeley. This routine includes the orthogonalization scheme of Gu and Eisenstat [12], the zero finder of Li [15] and is the subject of [16], where it is shown to be the *method of choice* for single processor computers. In this figure the line labeled **EQR** is the QR method, **EBZ** is bisection and inverse iteration [1], **EAR** is the new divide and conquer routine,

TRD is the tridiagonal reduction routine and **MM** is the matrix-matrix multiplication routine. The results in Fig. 6.2 are typical for the matrices which are reduced to tridiagonal form. With these matrices the new divide and conquer routine is between 1.25 to 4 times slower than the fastest routine depending on the test matrix.

In Fig. 6.3 the results are again normalized by the divide and conquer routine from the University of California at Berkeley. When a matrix is initially in tridiagonal form the routine from Berkeley can be much faster than the new routine. This type of performance is observed with arithmetically distributed eigenvalues as well as the clustered eigenvalue test problem. The performance of the other tridiagonal test matrices is similar to Fig. 6.2.

The parallel tests of this new routine were done on an Intel Paragon with 96 processors at the ETH. For the tests of the parallel routine two different sets of matrices were used. The first test matrix had twos on the principal diagonal and ones on the off diagonals. The speedup results for this matrix are given in Fig. 6.4. Here, speedup is

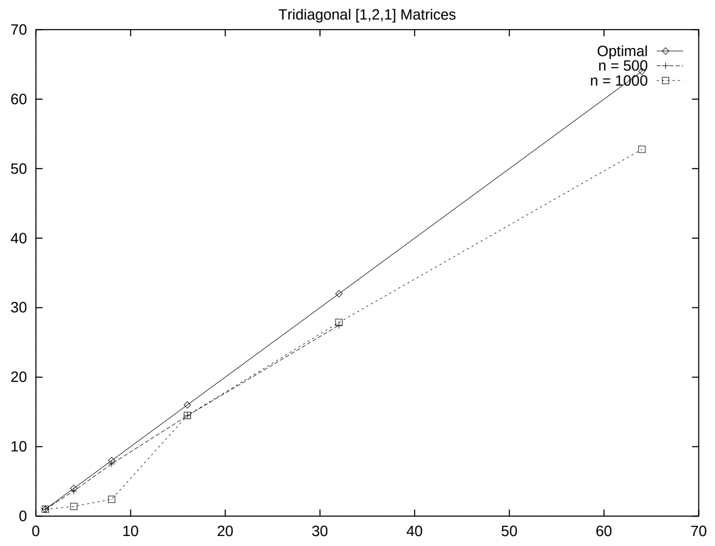

FIG. 6.4. *Speedup Results for* $[1, 2, 1]$ *Matrices*

defined as the time for the parallel routine **EAR** on p processors divided by the time for this routine on one processor. In this case **EAR** was 2.5 times slower than the fastest sequential routine, **EDC**. So, for the problem with $n = 1000$ and 64 processors a speedup of 21 was obtained relative to **EDC**.

The second test matrix was a random tridiagonal matrix. These matrices have a lot of deflation and thus are the severe test for the parallel routine. For this matrix the sequential routine was 2 times slower than the fastest sequential routine, which means that for the $n = 2000$ matrix with 64 processors a real speedup of 26 was observed. A graph of the speedup results for this matrix with sizes $n = 500, 1000, 2000$ is in Fig. 6.5.

7. Lessons Learned - Future Work.

After further investigation of the matrix multiplication routines it is evident that the implementation should be rewritten to use a two dimensional grid of processors. The matrix multiplication routines for a grid grow in communication with the square root of the number of processors, whereas this implementation has communication growth with the number of processors. In the

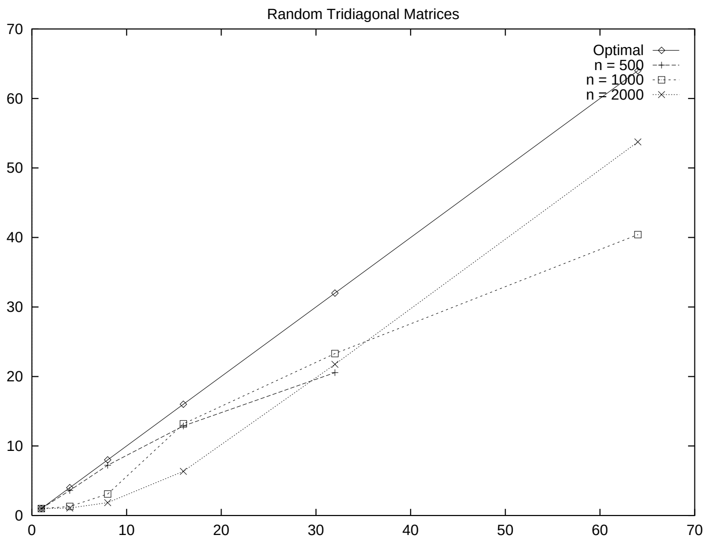

FIG. 6.5. *Speedup Results*

particular case of the Intel Paragon this decision had no bearing on the results since asynchronous communications could be effectively used to hide the communication costs for a large enough matrix. However, since communication is always slow with respect to floating point operations, and since larger messages have to be sent with the column scheme, the grid can effectively use a greater number of processors to solve any given problem.

Deflation is an important subject which has to be carefully considered and implemented. For the different types of deflation there are two ways of multiplying the Givens rotations. One can multiply the transpose of the Givens on the left by the eigenvector matrix of the outer matrix, or multiply it on the right by the resultant eigenvector matrix. Both of these options have been used with success. In order to gain the most efficient matrix multiplication, some of the Givens rotations should be multiplied to the left, or with the eigenvector matrices of the previous stage, and others rotations should be multiplied to the right, or with the eigenvector matrix of the current stage. To be specific, those Givens rotations which do not destroy the structure of the original eigenvector matrix should always to applied to that matrix. On the other hand, the Givens rotations which destroy the structure of the eigenvector matrix of the previous stage should multiply the newly found eigenvectors. In this way the maximum speedup for the matrix multiplication can be reached.

8. Conclusions.

Parallelization of the divide and conquer algorithm for symmetric eigenvalue problem on a distributed memory machine can deliver significant speedup results, while maintaining the accuracy and orthogonality of the eigenvector matrix. These routines should be considered for inclusion in any future collection of parallel routines for linear algebra.

A clear advantage for either the rank-one modification or the rank-one extension can not be drawn from the data available from these test results. It is possible to find test matrices where using rank-one modification formulation results in more deflation, on the other hand it should be possible to find matrices where the rank-one extension formulation results in more deflation, and is therefore faster. On the average both of the formulations have about the same performance and there are too many other factors including the choice of extended precision, the choice of a zero finder, the deflation criterion and the implementation of deflation which influence the timing results. These factors should be isolated and investigated in order to determine the appropriate divide and conquer algorithm. The appropriate divide and conquer algorithm may in fact be problem dependent.

As a parallel algorithm for the symmetric eigenvalue problem the divide and conquer algorithms have a definite advantage over the QR algorithm. Either a Gu-Eisenstat or an extended precision divide and conquer algorithm will be the method of choice for symmetric eigenvalue problems. At the current time a clear choice of which algorithm is the best can not be drawn.

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*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1992.

[2] P. Arbenz, *Divide and conquer algorithms for the bandsymmetric eigenvalue problem*, Parallel Computing, 18 (1992), pp. 1105–1128.

[3] P. Arbenz and K. Gates, *A note on divide and conquer algorithm for the symmetric tridiagonal eigenvalue problem*, Z. Angew. Math. Mech., 74 (1994), pp. T529–T531.

[4] J. Barlow and J. Demmel, *Computing accurate eigensystems of scaled diagonally dominant matrices*, SIAM J. Numer. Anal., 27 (1990), pp. 762–791.

[5] D. Boley and G. H. Golub, *A survey of matrix inverse eigenvalue problems*, Inverse Problems, 3 (1987), pp. 595–622.

[6] C. F. Borges and W. B. Gragg, *A parallel divide and conquer algorithm for the generalized real symmetric definite tridiagonal eigenvalue problem*, in Numerical Linear Algebra, L. Reichel, A. Ruttan, and R. S. Varga, eds., Berlin - New York, 1993, de Gruyter, pp. 11–29.

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

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

[9] T. J. Dekker, *A floating-point technique for extending the available precision*, Numer. Math., 18 (1971), pp. 224–242.

[10] J. W. Demmel, M. T. Heath, and H. A. van der Vorst, *Parallel numerical linear algebra*, Acta Numerica, (1993), pp. 111–198.

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

[12] M. Gu and S. C. Eisenstat, *A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem*, Research Report YALEU/DCS/RR-932, Department of Computer Science, Yale University, November 1992.

[13] ———, *A stable and efficient algorithm for the rank-one modification of the symmetric eigenproblem*, SIAM J. Matrix Anal. Appl., 15 (1994), pp. 1266–1276.

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

[15] R.-C. Li, *Solving secular equations stably and efficiently*. Unpublished Manuscript. Department of Mathematics, University of California, Berkeley CA, April 1993.

[16] J. Rutter, *A serial implementation of Cuppen's divide and conquer algorithm for the symmetric eigenvalue problem*, Tech. Report UCB/CSD 94/799, Computer Science Division, University of California, Berkeley CA, February 1994.

[17] 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.