Parallel Solution of Generalized Symmetric Tridiagonal Eigenvalue Problems on Shared Memory Multiprocessors

Calvin J. Ribbens and Christopher Beattie

TR 92-47

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

August 24, 1992

Parallel Solution of Generalized Symmetric Tridiagonal Eigenvalue Problems on Shared Memory Multiprocessors

Calvin J. Ribbens* Christopher Beattie†

Abstract

This paper describes and compares two methods for solving a generalized eigenvalue problem $Tx = \lambda Sx$, where T and S are both real symmetric and tridiagonal, and S is positive definite, and the target architecture is a shared memory multiprocessor. One method can be viewed as a generalization of the *treeql* algorithm of Dongarra and Sorensen [10]. The second algorithm is a straightforward parallel extension of the bisection/inverse iteration algorithm *treps* of Lo, Philippe, and Sameh [21]. The two methods are representative of families of algorithms of quite different character. We illustrate and compare sequential and parallel performance of the two approaches with numerical examples.

1 Introduction

This paper describes and compares two parallel algorithms for solving a generalized eigenvalue problem

$$T\mathbf{x} = \lambda S\mathbf{x}, \tag{1}$$

where T and S are both symmetric and tridiagonal, and S is positive definite. Both of the algorithms considered here are generalizations of well known algorithms for the standard symmetric tridiagonal eigenvalue problem. In this section we briefly describe an application where a problem such as (1) arises and survey related work in the literature.

Problems having such structure occur in higher order difference approximations of Sturm-Liouville and radial Schrödinger equations. The approximation of eigenvalues and eigenfunctions of such operators is a common problem in computational physics and often functions as a starting point for complex multidimensional problems. As such there is a natural interest in such calculations being done efficiently and to reasonably high accuracy. Finite difference methods are reliable and efficient in this setting and high accuracy formulations have appeared under various names such as Mehrstellen methods [7], operator implicit methods [4], and HODIE methods [5]. Recent work in this direction may also be found within the physics community [14].

The work reported in this paper draws heavily from two algorithms designed for the standard symmetric tridiagonal eigenvalue problem. The first is a divide-and-conquer scheme, *treeql*, due to Dongarra and Sorensen [10]. The second algorithm is *treps*, described by Lo, Philippe, and Sameh [21], and based on bisection and inverse iteration. There are other families of methods

*Department of Computer Science, Virginia Polytechnic Institute & State University, Blacksburg, VA 24061, ribbens@cs.vt.edu, (fax) 703/231-6075. Research supported by Department of Energy grant DE-FG05-88ER25068.

†Department of Mathematics, Virginia Polytechnic Institute & State University, Blacksburg, VA 24061. Research supported by the National Science Foundation under grant NSF 0647-5407. A portion of this work was done while this author was a fellow of the Alexander von Humboldt Foundation.

which are of interest for this problem (e.g., Lanczos methods, Jacobi methods, homotopy methods, etc.). Related references include Li and Rhee [19], Li, Zhang, and Sun [20], and Weston, Clint, and Bleakney [28]. It is only generalizations of the two algorithms *treps* and *treeql* that are compared in this paper.

The *treeql* algorithm is based on a divide-and-conquer scheme proposed initially by Cuppen [9], and uses results on rank-one updates and a special root-finding algorithm from Bunch, Nielsen, and Sorensen [6]. Experimental results with *treeql* are given in [10] for shared memory machines. Jessup [16] and Ipsen and Jessup [15] consider both divide-and-conquer and bisection approaches for distributed memory architectures. Important details regarding the accuracy of divide-and-conquer schemes have been studied by Barlow [1] and Sorensen and Tang [26]. Gates [12] uses inverse iteration to improve the performance of a divide-and-conquer algorithm. A different application of the divide-and-conquer idea to the standard symmetric eigenvalue problem is described by Krishnakumar and Morf [18].

There are few references in the literature on efficient parallel algorithms for the generalized symmetric eigenvalue problem. In the most common applications, T and S are banded (S positive definite) but not necessarily tridiagonal. Crawford [8] describes an efficient algorithm for reducing such a problem to an ordinary symmetric (banded) problem. Kaufman [17] has proposed a vectorized version of Crawford's algorithm. Ma, Patrick, and Szyld [22] and Pantazis and Szyld [24] have described algorithms for the banded generalized problem which are based on bisection and inverse (Rayleigh quotient) iteration. They also report results for shared memory multiprocessors. Shougen and Shuqin [25] have described an algorithm for the banded case which is based on a generalized singular value decomposition and requires both S and T to be symmetric and positive definite. The case were T and S are dense is considered by Natarajan [23].

The rest of the paper is organized as follows. Generalized versions of *treeql* and *treps* are described in §3 and 2, respectively. In §4 we use several numerical examples to illustrate and compare the performance of the two algorithms. Finally, we conclude with some comments and suggestions for further research in §5.

2 A divide-and-conquer approach

In this section we first summarize the theoretical framework of a divide-and-conquer scheme for generalized symmetric tridiagonal eigenvalue problems, and then describe our algorithm and its parallel implementation. See [3] for a complete discussion of the theoretical framework of the algorithm and its performance, including error analysis.

2.1 Theoretical framework

The critical focus in generalizing the standard divide-and-conquer scheme involves dividing the task of diagonalizing a tridiagonal matrix pencil, $T - \lambda S$, into subsidiary tasks that involve diagonalizing two tridiagonal matrix pencils of roughly half the size. For example, we define subproblems of size k and $n-k$ by rewriting T and S as

$$T = \begin{bmatrix} T_1 & \alpha \mathbf{e}_k \mathbf{e}_1^t \\ \alpha \mathbf{e}_1 \mathbf{e}_k^t & T_2 \end{bmatrix} = \begin{bmatrix} \hat{T}_1 & 0 \\ 0 & \hat{T}_2 \end{bmatrix} + \theta_1 \alpha \begin{bmatrix} \mathbf{e}_k \\ \theta_1^{-1} \mathbf{e}_1 \end{bmatrix} \begin{bmatrix} \mathbf{e}_k^t & \theta_1^{-1} \mathbf{e}_1^t \end{bmatrix}, \quad \text{and} \qquad (2)$$

$$S = \begin{bmatrix} S_1 & \beta \mathbf{e}_k \mathbf{e}_1^t \\ \beta \mathbf{e}_1 \mathbf{e}_k^t & S_2 \end{bmatrix} = \begin{bmatrix} \hat{S}_1 & 0 \\ 0 & \hat{S}_2 \end{bmatrix} + \theta_2 \beta \begin{bmatrix} \mathbf{e}_k \\ \theta_2^{-1} \mathbf{e}_1 \end{bmatrix} \begin{bmatrix} \mathbf{e}_k^t & \theta_2^{-1} \mathbf{e}_1^t \end{bmatrix}, \qquad (3)$$

where θ_1 and θ_2 can be selected independently to minimize cancellation in the affected diagonal entries as in [10].

Suppose now that $\hat{S}_1$ and $\hat{S}_2$ as defined in (3) are positive-definite and we have solved the two smaller generalized eigenvalue problems $\hat{T}_1\hat{X}_1 = \hat{S}_1\hat{X}_1\hat{D}_1$ and $\hat{T}_2\hat{X}_2 = \hat{S}_2\hat{X}_2\hat{D}_2$, where $\hat{X}_1, \hat{X}_2$ are matrices of eigenvectors for the respective systems (normalized so that $\hat{X}_i^t\hat{S}_i\hat{X}_i = I$) and $\hat{D}_1, \hat{D}_2$ are diagonal matrices containing eigenvalues of the respective systems. If we then denote $\hat{X} = \text{diag}(\hat{X}_1, \hat{X}_2)$, $\hat{D} = \text{diag}(\hat{D}_1, \hat{D}_2)$, $\mathbf{y}_1 = \hat{X}_1^t\mathbf{e}_k$, and $\mathbf{y}_2 = \hat{X}_2^t\mathbf{e}_1$, we find

$$\hat{X}^t(T - \lambda S)\hat{X} = \\ \hat{D} - \lambda I + \theta_1\alpha \begin{bmatrix} \mathbf{y}_1 \\ \theta_1^{-1}\mathbf{y}_2 \end{bmatrix} \begin{bmatrix} \mathbf{y}_1^t & \theta_1^{-1}\mathbf{y}_2^t \end{bmatrix} - \lambda\theta_2\beta \begin{bmatrix} \mathbf{y}_1 \\ \theta_2^{-1}\mathbf{y}_2 \end{bmatrix} \begin{bmatrix} \mathbf{y}_1^t & \theta_2^{-1}\mathbf{y}_2^t \end{bmatrix}. \tag{4}$$

Note that $\mathbf{y}_1^t$ is the bottom row of $\hat{X}_1$ and $\mathbf{y}_2^t$ is the top row of $\hat{X}_2$.

The main task remaining is to diagonalize the right hand side of (4), since if X and $D = \text{diag}(\delta_1, \ldots, \delta_n)$ contain the eigenvectors and eigenvalues, respectively, for (4), then $Y \equiv \hat{X}X$ and D satisfy $Y^tTY = D$ and $Y^tSY = I$, as desired. If $\beta = 0$ then the problem reduces immediately to calculating a rank-one update as in [9] and [10]. It is in the case $\beta \neq 0$ that we have our most significant deviation from the Cuppen-Dongarra-Sorensen line of development. One may indeed follow their pattern of reasoning to determine how to complete updates arising from the original tearings—however notice that for independently determined values of θ_1 and θ_2, (4) represents a rank-two change on the unperturbed pencil $\hat{D} - \lambda I$. This has grave consequences for the ensuing root-finding problem, since it will no longer be true in general that the eigenvalues of (4) interlace the eigenvalues of $\hat{D}$. Fortunately, we may recover most of what has been lost with a slight sacrifice of generality.

Assume henceforth that $\beta \neq 0$ without loss of generality. The rank-two modification problem implicit in (4) may be mapped to an equivalent rank-one *restriction* problem if the ranges of the twin rank-one tearings are identical. This has an important consequence, namely the interlacing of perturbed and unperturbed eigenvalues, and so leads to a root finding problem that is very similar to what is found in [6]. The identification of ranges occurs if and only if $\theta_1 = \theta_2$. This may raise some concern, since θ_1 and θ_2 were chosen at least in part to guard against cancellation errors during tearing. However, the analysis of *treeql* by Barlow [1] suggests that the values of θ_1 and θ_2 are not likely to be critical in our situation, at least in circumstances where S is "safely" positive-definite. This is borne out in the analysis of [3] (following Barlow's line of argument) and is consistent with our computational experience.

We shall assume henceforth that $\theta_1 = \theta_2 = \theta = \pm 1$ and consider in detail the beneficial consequences. The right-hand side of (4) becomes

$$\hat{D} - \lambda I + \theta(\alpha - \lambda\beta)\mathbf{z}\mathbf{z}^t = [I \quad \mathbf{z}] \left(\begin{bmatrix} \hat{D} & 0 \\ 0 & \theta\alpha \end{bmatrix} - \lambda \begin{bmatrix} I & 0 \\ 0 & \theta\beta \end{bmatrix} \right) \begin{bmatrix} I \\ \mathbf{z}^t \end{bmatrix} \tag{5}$$

where $\mathbf{z}^t = \begin{bmatrix} \mathbf{y}_1^t & \theta^{-1}\mathbf{y}_2^t \end{bmatrix}$. The interior matrix pencil on the right-hand side of (5) is definite if $\theta\beta > 0$, hence we take $\theta = sgn(\beta)$.

Now absorb $\theta\beta = |\beta|$ into the flanking matrices and define $\bar{\mathbf{z}} = \sqrt{|\beta|}\mathbf{z}$ and $\mathcal{P} = \text{Ran}\,[I \quad \bar{\mathbf{z}}]^t$, an n-dimensional subspace of $\Re^{n+1}$. For $\bar{D} = \text{diag}(\hat{D}, \alpha/\beta)$, the eigenvalues of (5) are precisely the stationary values of the restricted Rayleigh quotient

$$\left.\frac{\mathbf{u}^t\bar{D}\mathbf{u}}{\mathbf{u}^t\mathbf{u}}\right|_{\mathbf{u}\in\mathcal{P}}. \tag{6}$$

In this way the eigenvalues of $T - \lambda S$ may be associated with the eigenvalues of a restricted matrix eigenvalue problem (cf. [13] and [2]). The following is proved in [3].

Theorem 1 *If $\hat{D} = diag(\hat{\delta}_i)$ has only simple eigenvalues and no component of $\mathbf{z}$ is zero then either*

(1) $\mu = \alpha/\beta$ is an eigenvalue of $T - \lambda S$ and $\alpha/\beta = \hat{\delta}_k$ for some k, or

(2) $\bar{D}$ has only simple eigenvalues and $\bar{D} - \mu I$ is nonsingular for every eigenvalue μ of $T - \lambda S$.

Deflation may be used as in [6] to eliminate multiple eigenvalues of $\hat{D}$ and zero entries of $\mathbf{z}$, thus ensuring that the hypotheses of the theorem always hold. The first case of the conclusion corresponds to a further deflation step which is taken if α/β is within a tiny tolerance of an eigenvalue of $\hat{D}$. The second case of the conclusion represents the fully deflated problem and leads to a secular equation as described in the next paragraph.

Notice that $\mathcal{P}^{\perp} = \text{span}\,[\bar{\mathbf{z}}^t \quad -1]^t$. Every eigenpair $(\mu, \mathbf{u})$ of (6) satisfies

$$(\bar{D} - \mu I)\mathbf{u} = \sigma \begin{bmatrix} \bar{\mathbf{z}} \\ -1 \end{bmatrix}$$

for some scalar σ. If the second case of Theorem 1 holds then $\sigma \neq 0$ and

$$\mathbf{u} = \sigma(\bar{D} - \mu I)^{-1} \begin{bmatrix} \bar{\mathbf{z}} \\ -1 \end{bmatrix}. \tag{7}$$

Since $[\bar{\mathbf{z}}^t \quad -1]\,\mathbf{u} = 0$, μ must satisfy

$$w(\mu) = \sum_{i=1}^{n+1} \frac{\zeta_i^2}{\bar{\delta}_i - \mu} = 0. \tag{8}$$

In (8), $\{\bar{\delta}_i\}_{i=1}^{n+1}$ list the increasingly ordered entries of $\bar{D}$ and $\{\zeta_i\}_{i=1}^{n+1}$ list the entries of $\bar{\mathbf{z}}$ together with 1 ordered consistently with the ordering of the $\bar{\delta}_i$'s. Equation (8) is precisely the secular equation given by Golub [13] for a rank-one restriction problem. Notice that the function $w(\lambda)$ has many of the same qualitative features as its counterpart in [6] (e.g., root/singularity interlacing, monotonicity) and so, the sophisticated root-finding scheme of [6] may be used with minor alterations to find an eigenvalue of (5). Finally, given an eigenvalue μ of (5) we know from Theorem 5 of [6] that an eigenvector $\mathbf{x}$ is given by

$$\mathbf{x} = \gamma(\hat{D} - \mu I)^{-1}\mathbf{z}, \tag{9}$$

where γ is chosen to normalize $\mathbf{x}$ with respect to $I + \theta\beta\mathbf{z}\mathbf{z}^t$.

Just as for divide-and-conquer methods for the standard tridiagonal eigenvalue problem, considerable savings can be realized in many circumstances from deflation, i.e., when an eigenpair is inherited for almost no work from the unperturbed problem $D - \lambda I$. The two ways in which this may occur in the standard case have direct analogs in the generalized case. For example, it is straightforward to see that if $\mathbf{z}^t\mathbf{e}_j = 0$ then $(\hat{d}_j, \mathbf{e}_j)$, the ith eigenpair of $\hat{D} - \lambda I$, is also an eigenpair of (5). Furthermore, one can easily show that if $\hat{d}_j$ is an eigenvalue of $\hat{D}$ of multiplicity $r \geq 2$ then orthogonal transformations may be applied to (5) to introduce $r - 1$ zeros in $\mathbf{z}$, and thus reduce to the first type of deflation. In the generalized case we have one further possibility for deflation, as indicated by the first case of Theorem 1. If $\alpha/\beta = \hat{d}_j$, for some $\hat{d}_j \in \lambda(\hat{D})$, then $(\hat{d}_j, \mathbf{e}_j)$ is an eigenpair of (5).

See [3] for an extensive discussion of important details regarding error analysis, deflation criteria in finite precision, and the algorithm for finding roots of the secular equation (8).

Table 1: Time for *rsg* and *crw*, and their ratio, with and without eigenvector calculations on Problem 1.

	Full Computation			Eigenvalues Only		
n	*rsg*	*crw*	Ratio	*rsg*	*crw*	Ratio
32	1.21	0.98	1.23	0.42	0.15	2.80
64	9.76	6.98	1.40	3.31	0.57	5.81
128	78.13	51.78	1.51	27.41	2.24	12.24
256	620.47	396.74	1.56	221.54	8.84	25.06

2.2 The generalized algorithm

Given the matrix tearing scheme defined by (5), it is relatively easy to see how to apply the strategy recursively to derive an efficient parallel algorithm. Our algorithm, *gtreeql*, is most easily described as a generalization of the *treeql* algorithm of Dongarra and Sorensen [10]. A binary tree represents the various levels of decomposition: the root corresponds to the first tear into two subproblems, the nodes at the next level represent a split into four subproblems, etc. Each leaf of the tree represents two small subproblems which are solved independently, and then combined using the updating procedure. Thus, each node of the tree represents one rank-one update process, with nodes at higher levels representing larger subproblems. The updating procedure consists of checking for and exploiting deflation, computing the eigenvalue of the deflated problem by finding the roots of the secular equation (8), and computing the new eigenvectors using (9). A parameter, *nsmall*, determines the size of the problems that will be solved at a leaf node. Subproblems are recursively split until the pieces are at most *nsmall* $\times$ *nsmall*.

An efficient solution method for the individual subproblems at the leaves is an important component of *gtreeql*. The straightforward (EISPACK) approach would probably be to use *rsg*, which transforms the problem to a dense standard problem, reduces to tridiagonal standard form using Householder transformations, and then diagonalizes using QL iterations (*tql2*). A better strategy is to use a reduction scheme described by Crawford [8] to first transform the problem to an ordinary symmetric tridiagonal problem, and then use *tql2* to diagonalize (or *tql1* if only eigenvalues are desired). Crawford's algorithm uses elementary congruence transformations on T and S to reduce S to the identity matrix, and orthogonal similarity transformations to "chase" down the diagonal the fill which results in T, thus preserving the bandedness of T. Table 1 illustrates the superiority of this approach. Crawford's algorithm followed by *tql1* (column *crw* in Table 1) is an $O(n^2)$ process if eigenvectors are not required, while *rsg* is $O(n^3)$. Even if eigenvectors are desired, experiments indicate that *crw* (in this case, Crawford reduction followed by *tql2*) is approximately 50% faster. There are also considerable savings in storage in using *crw* instead of *rsg*.

The basic structure of the update procedure and the root-finder from *treeql* remains the same in *gtreeql*. We observe the same convergence behavior in the root-finder as described in [10]; i.e., given an initial guess to the left of the root, convergence is monotonic and asymptotically quadratic. Choosing an initial guess and solving for a new estimate of the root at each step are also straightforward to generalize.

We also made a few modifications to the *treeql* code which are not directly related to the generalized problem. In particular, we modified the rank-one update procedure to exploit the block-diagonal structure of the matrix of eigenvectors. This results in time savings of nearly a factor of two. We also implemented a version of *gtreeql* that computes eigenvalues only. In order to carry out the rank-one updating procedure in this case, it is still necessary to compute the first and last components of the eigenvectors. This can be done cheaply in the updating step, but we must carry out the full eigenvector calculation for the smallest subproblems. It is clear that using only a few levels of decomposition is a mistake for the eigenvalue-only case, since the subproblems at the leaves would still be relatively large. Finally, we mention that for the eigenvalue-only case it is best to replace the general $O(n^2)$ eigenvalue sorting routine in *treeql* by an $O(n)$ merge routine, again exploiting the special structure of the divide-and-conquer setting.

2.3 Parallel implementation

Our parallel implementation of *gtreeql* uses the SCHEDULE package [11] in exactly the same way as *treeql* does. SCHEDULE is a set of FORTRAN-callable routines which allows programmers to define parallel tasks and a control flow graph indicating which tasks may execute when. The SCHEDULE software maintains a queue of the tasks ready to execute, and assigns these tasks to processors as they become available. While the basic control flow graph must be defined statically (i.e., before user-defined tasks begin execution), SCHEDULE does allow some dynamic spawning of tasks as well. SCHEDULE has been implemented on many shared memory architectures, and hence provides some portability for parallel programs.

The static control flow graph for *gtreeql* is the binary tree described above. Each node represents a single rank-one update process in which the solutions to two subproblems are combined into the solution to a problem of roughly twice the size. An update task may proceed as soon as its two immediate descendents in the tree have completed; it is completely independent of any other update node in the tree. Recall that a leaf node corresponds to an update also, but that its two subproblems are solved directly by *crw*. These *crw* solves at the leaves are independent tasks as well. Finally, during a rank-one update each eigenpair of the larger problem may be computed independently. This relatively fine-grained parallelism is only exploited, however, if there are not enough rank-one update steps available to keep all processors busy. As we move up the tree, this will clearly be the case, since there are fewer and fewer update nodes at higher levels in the tree. Thus, if granularity requirements warrant it, the eigenpair computations—a root-solve and a few matrix-vector operations to compute the eigenvector—are dynamically spawned by the update process as further independent tasks. In summary, parallelism in *gtreeql* is exploited primarily in three steps:

1. Independent *crw* solves at the leaves.

2. Independent rank-one update computations at each node.

3. Within a rank-one update step, independent eigenpair calculations.

3 An approach based on bisection and inverse iteration

We describe *gtreps*, a generalization of *treps*. Although the work of Szyld and colleagues [22, 24] represents a more sophisticated generalization of *treps* in many respects, our relatively straightforward generalization is sufficient for making good comparisons with other approaches. We proceed by first reviewing *treps* itself and then describing the modifications required for *gtreps* and discussing its parallel implementation.

3.1 Review of *treps*

The *treps* algorithm proposed by Lo, Philippe, and Sameh [21] computes the eigenvalues in a given interval, and associated eigenvectors, of a symmetric tridiagonal matrix T. It can be viewed as a variant of the EISPACK routine *tsturm*, since it is based on bisection and inverse iteration. In *treps*, multisection is used rather than bisection, and there is some re-organization of the code to enhance vectorization and parallelization. The results reported in [21] from experiments on two vector multiprocessors (an Alliant FX/8 and a CRAY X-MP/48) indicate that *treps* can be a very efficient algorithm on such architectures.

For completeness, and in anticipation of our discussion of the generalized algorithm below, we list the four basic steps in *treps*:

1. Isolation of eigenvalue clusters by multisection.
2. Extraction of an eigenvalue cluster by bisection (*treps1*) or the Zeroin method (*treps2*).
3. Computation of eigenvectors by inverse iteration.
4. Orthogonalization of eigenvectors in a group by the Modified Gram-Schmidt process.

A cluster is one or more computationally coincident eigenvalue. In the case of a computationally simple eigenvalue, Step 1 computes an interval known to contain only that eigenvalue. Step 2 then computes the eigenvalue to the required precision. If a cluster contains more than one eigenvalue, Step 1 generally stops only when the interval containing the cluster is so small that Step 2 is skipped. The only difference between the two algorithm variants, *treps1* and *treps2*, is the root-finding scheme used for extracting eigenvalues in Step 2. The groups of eigenvectors mentioned in Step 4 are defined in terms of their associated eigenvalues. Let $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$ be the eigenvalues of T. Then $\lambda_k, \ldots, \lambda_l$ form a group if $k \leq l$ and

$$\lambda_{j+1} - \lambda_j < \varepsilon \equiv 10^{-3} \max_{1 \leq i \leq n} \left(|t_{ii}| + |t_{i-1,i}|\right), \text{ for } j = k, \ldots, l-1,$$

but $\lambda_{l+1} - \lambda_l \geq \varepsilon$ and $\lambda_k - \lambda_{k-1} \geq \varepsilon$. In other words, two adjacent eigenvalues are placed in different groups if they have a gap of at least ε separating them. Since the Gram-Schmidt process is only run on eigenvectors within a group, the size of these groups (or, equivalently, the degree of separation among the eigenvalues) has a tremendous impact on the overall cost of the algorithm.

3.2 The generalized algorithm

Modifying *treps* for the generalized symmetric tridiagonal eigenvalue problem $Tx = \lambda Sx$ is fairly straightforward. The main ingredients of the algorithm—Sturm sequences, multisection, inverse iteration, Gram Schmidt orthogonalization—all have direct analogs in the generalized case. As with *treps*, there are two versions of the algorithm, depending on how eigenvalues are extracted: *gtreps1* uses bisection and *gtreps2* uses Zeroin.

If $p_n(\lambda) = \det(T - \lambda S)$, then it is well known that the Sturm sequence defined by

$$\begin{aligned} p_0(\lambda) &= 1 \\ p_1(\lambda) &= t_{11} - \lambda s_{11} \\ p_i(\lambda) &= (t_{ii} - \lambda s_{ii})p_{i-1}(\lambda) - (t_{i-1,i} - \lambda s_{i-1,i})^2 p_{i-2}(\lambda), \quad i = 2, \ldots n, \end{aligned} \tag{10}$$

may be used to evaluate $p_n(\lambda)$, and furthermore, that the number of sign changes in the sequence $\{p_i(\lambda)\}$ is equal to the number of eigenvalues smaller than λ. It is also well known that the linear recurrence (10) is more likely to suffer from overflow than the sequence

$$q_i(\lambda) = t_{ii} - \lambda s_{ii} - \frac{(t_{i-1,i} - \lambda s_{i-1,i})^2}{q_{i-1}(\lambda)}, \quad i = 2, \ldots, n, \tag{11}$$

where

$$q_i(\lambda) = \frac{p_i(\lambda)}{p_{i-1}(\lambda)}, \quad i = 1, \ldots, n.$$

Hence, the nonlinear recurrence (11) is usually used in bisection algorithms, with the number of negative terms in the sequence $\{q_i(\lambda)\}$ giving the number of eigenvalues less than λ. Note however that in order to use a root finder such as Zeroin to extract an isolated eigenvalue, the recurrence (10) must be used because one needs a function $p_n(\lambda)$ that is zero at the root. If bisection is used to extract λ then the recurrence (11) may be used throughout. Note also that the term $(t_{i-1,i} - \lambda s_{i-1,i})^2$ in both (10) and (11) must be re-computed for each λ. In the standard eigenvalue case, the analogous term is simply $t_{i-1,i}^2$ and thus can be computed once and for all and saved in an extra n-vector.

As stated, the major components of *treps* remain the same in our generalized algorithm. We follow exactly the four steps listed in §3.1. The grouping criterion used by [21] is ad hoc though effective and seeks to group eigenvalues together if a perturbation of the matrix roughly of order $10^{-3}\|T\|$ could bring the eigenvalues together. Following this idea, we choose to group eigenvalues together if a perturbation of T on the order of $10^{-3}\|T\|\;\|S\|$ or less could bring the eigenvalues together. So, λ_i and λ_j will be members of the same group if $|\lambda_i - \lambda_j| \leq \|S^{-1/2}ES^{-1/2}\|$ where $\|E\| \leq 10^{-3}\|T\|\;\|S\|$. This is implied by the criterion $|\lambda_i - \lambda_j| \leq 10^{-3}\|T\|\text{cond}_2(S)$. In our code we actually use Gershgorin bounds for $\text{cond}_2(S)$ and a cheap estimate for $\|T\|$ (similar to [21]).

3.3 Parallel implementation

The original *treps* algorithm was developed for vector multiprocessors, so considerable attention was paid to vectorization issues. Since our first target machine is a shared memory multiprocessor without vector capability, our parallelization of *gtreps* is somewhat simpler. Vectorization could be added in a manner completely analogous to what is done in *treps*. We exploit parallelism in three ways in *gtreps*:

1. Independent Sturm sequences are evaluated in parallel during a multisection step. Each multisection step generally divides a given interval into $P + 1$ subintervals (requiring P new Sturm sequences), where P is the number of available processors. Each resulting subinterval that still contains more than one eigenvalue (and has not collapsed into a cluster of coincident eigenvalues) is multisectioned again.

2. Extraction of a single eigenvalue and computation of an associated eigenvector by inverse iteration is a single independent task. Obviously, these can be done for different eigenvalues at the same time.

3. The orthogonalization step for a group of m eigenvectors requires $O(mn^2)$ operations. The dominant loops can easily be parallelized so that the granularity of a single task is $O(mn/P)$.

4 Numerical experiments

In this section we give numerical results to illustrate the performance of *gtreeql*, *gtreps1*, and *gtreps2*. In addition to parallel performance on a single large problem, we report the accuracy and sequential performance for each method as problem size varies. For the purposes of comparison, sequential data for *crw* is also included.

Three test problems $T\mathbf{x} = \lambda S\mathbf{x}$ are used. Problem 1 is a model problem (e.g., representing a simple finite element model for longitudinal vibration of an elastic bar), with $T = \text{tridiag}(-1, 2, -1)$ and $S = \text{tridiag}(1, 4, 1)$. Problem 2 is a perturbation of Problem 1, with the main diagonals of T and S, and the off-diagonals of T, perturbed by a random variable uniformly distributed in $[-.5, .5]$. Problem 3 uses a random T with t_{ii} and $t_{i-1,i}$ drawn from a uniform distribution in $[-1, 1]$, and $S = \text{tridag}(1/4, 1, 1/4)$. Most of the data reported is based on the first two test problems. In Section 4.3 we report performance on up to 100 different cases of of Problem 3.

Both methods we consider here construct explicit eigenvector bases and hence are sensitive to the conditioning of the right hand matrix S. In order to avoid spurious ill-conditioning due to scaling, we first diagonally scale the matrix pencil $T - \lambda S$ so that $s_{ii} = 1$. This guarantees that $\text{cond}_2(S)$ is within a factor of 2 of the minimal possible over all diagonal scalings of S [27]. In turn this would yield a final eigenvector basis that is nearly as well-conditioned as is feasible over all diagonal scalings of the original matrix pencil.

All the experiments reported here were performed on a Sequent Symmetry S81. Double precision, under the ATS FORTRAN compiler, was used throughout. All times reported in the tables below are in seconds, and each represents the average of at least three runs. Variation from run to run was small: never more than 5%. The *gtreeql* parameter *nsmall* is set to 32 unless otherwise indicated. For *gtreps1* and *gtreps2* we force all eigenvalues to be in a single group for the purpose of orthogonalization, unless otherwise indicated. Section 4.5 looks specifically at the issue of orthogonalization in *gtreps*.

The remainder of this section is organized as follows. Section 4.1 summarizes the accuracy of each method, reporting measures of the size of the residual and of the orthogonality of the eigenvectors. Sequential performance is discussed in §4.2, and parallel performance of the basic methods is reported in §4.4. Finally, in §§4.5–4.7 we discuss three important issues which can have a significant effect on the performance of these algorithms.

4.1 Accuracy

It is important that accuracy—measured in the residual and in orthogonality of the eigenvectors—be preserved as problem size grows. Tables 2 and 3 show that both measures of accuracy are quite good for the problem sizes considered. There does appear to be a slight loss of orthogonality with *gtreeql* and *crw*. We are currently investigating ways to improve the orthogonality of the eigenvectors computed by *gtreeql* (e.g., by improving deflation criteria and the accuracy of the root solver). Note that orthogonality for the *gtreps* methods is very good, but this depends entirely on the orthogonalization step mentioned in §3. Below (in §4.5) we consider the effect of changing the logic used in deciding which eigenvectors belong in the same group. Finally, we remark that without diagonal scaling *gtreps2* fails for the largest case of Problem 2. As mentioned above, the linear recurrence on which *gtreps2* is based suffers from the possibility of overflow, and that is exactly what happens in this case. With scaling we can avoid overflow for this problem, but it is a concern for other problems.

Table 2: Residual, $\max_i ||T\mathbf{x}_i - \lambda_i S\mathbf{x}_i||_2$.

	n	*crw*	*gtreeql*	*gtreps1*	*gtreps2*
Problem 1	32	9.0×10^{-15}	9.0×10^{-15}	4.5×10^{-14}	3.8×10^{-15}
	64	1.4×10^{-14}	7.9×10^{-15}	2.2×10^{-14}	1.0×10^{-14}
	128	2.4×10^{-14}	8.6×10^{-15}	5.2×10^{-13}	1.2×10^{-14}
	256	4.0×10^{-14}	9.4×10^{-15}	7.0×10^{-13}	3.0×10^{-14}
	512	7.3×10^{-14}	9.8×10^{-15}	1.3×10^{-12}	1.9×10^{-12}
Problem 2	32	7.5×10^{-15}	7.5×10^{-15}	1.1×10^{-14}	3.2×10^{-15}
	64	1.3×10^{-14}	1.1×10^{-14}	2.1×10^{-14}	4.0×10^{-15}
	128	2.4×10^{-14}	1.2×10^{-14}	5.3×10^{-14}	8.0×10^{-15}
	256	5.3×10^{-14}	1.2×10^{-14}	1.7×10^{-13}	1.4×10^{-14}
	512	8.2×10^{-14}	2.1×10^{-14}	9.0×10^{-13}	1.0×10^{-12}

Table 3: Orthogonality, $\max_{i,j} |\mathbf{x}_i^t S\mathbf{x}_j - \delta_{ij}|$.

	n	*crw*	*gtreeql*	*gtreps1*	*gtreps2*
Problem 1	32	6.4×10^{-15}	6.4×10^{-15}	6.7×10^{-16}	6.7×10^{-16}
	64	1.2×10^{-14}	1.8×10^{-14}	1.1×10^{-15}	6.7×10^{-16}
	128	2.0×10^{-14}	3.6×10^{-14}	1.1×10^{-15}	1.7×10^{-15}
	256	3.8×10^{-14}	2.5×10^{-13}	1.6×10^{-15}	1.7×10^{-15}
	512	7.1×10^{-14}	5.9×10^{-13}	3.1×10^{-15}	2.4×10^{-15}
Problem 2	32	3.3×10^{-15}	3.3×10^{-15}	6.7×10^{-16}	4.4×10^{-16}
	64	9.8×10^{-15}	6.7×10^{-15}	6.7×10^{-16}	6.7×10^{-16}
	128	1.8×10^{-14}	1.4×10^{-14}	1.4×10^{-15}	1.6×10^{-15}
	256	3.8×10^{-14}	1.3×10^{-14}	2.0×10^{-15}	1.7×10^{-15}
	512	5.7×10^{-14}	8.2×10^{-14}	4.2×10^{-15}	3.1×10^{-15}

Table 4: Sequential performance: time and speedup over *crw*.

	n	*crw*	*gtreeql*		*gtreps1*		*gtreps2*	
Problem 1	32	0.98	1.02	1.0	1.50	0.7	0.63	1.6
	64	6.98	2.70	2.6	6.90	1.0	3.56	2.0
	128	51.78	11.55	4.5	37.08	1.4	24.28	2.1
	256	396.74	60.49	6.6	228.02	1.7	179.35	2.2
	512	3051.32	390.44	7.8	1586.16	1.9	1374.09	2.2
Problem 2	32	0.99	1.02	1.0	1.49	0.7	0.65	1.5
	64	6.88	3.72	1.8	6.85	1.0	3.65	1.9
	128	50.99	16.91	3.0	36.96	1.4	24.73	2.1
	256	392.35	89.17	4.4	227.89	1.7	180.41	2.2
	512	3044.46	509.37	6.0	1569.31	1.9	1375.21	2.2

4.2 Sequential performance

When comparing two parallel algorithms it is important to know how much of the advantage of one over the other is due to parallelization and how much is simply due to better sequential performance. Hence, before turning to parallel results, we briefly discuss performance on a single processor.

The data in Table 4 indicate a strong advantage for *gtreeql* over the other methods when all the eigenvalues and eigenvectors are computed. The advantage is greater for Problem 1 than for Problem 2 because of the significant amount of deflation occurring in this problem. For example, when $n = 512$ there are 549 deflations with Problem 1 but only 124 with Problem 2. Recall that each time a deflation occurs, an eigenpair is computed at essentially no cost. The other three methods take no advantage of deflation, so their results are essentially the same for the two problems. Even without substantial deflation, the computational complexity of *gtreeql* appears to be something less than $O(n^3)$, so that its advantage over the other methods increases with n. This parallels the results obtained by Dongarra and Sorensen with *treeql*, where as a sequential algorithm it outperformed *tql2*.

It must be pointed out that the time required by *gtreps1* and *gtreps2* is dominated by the orthogonalization step as n grows. For example, on Problem 1 with n = 128, 256, and 512, the percentage of time spent in orthogonalization is 23.7%, 66.1%, and 94.5%, respectively. We comment briefly on the effect of changing orthogonalization strategies in §4.5 below. Note also that the performance of *gtreeql* can be sensitive to *nsmall*, the size of the leaf problems. This value is fixed at 32 for the data in Table 4, which means that no splitting occurs at all for the smallest case (i.e., *gtreeql* reduces to *crw* with some extra overhead). Varying *nsmall* can improve the performance of *gtreeql* slightly for small n. We comment on the effect of changing *nsmall* in §4.7.

Turning to the case where only the eigenvalues are computed, we see from Table 5 that *crw* is consistently the fastest sequential method. It enjoys an advantage of at least a factor of two over *gtreeql* and of nearly ten over *gtreps1*; *gtreps2* is somewhat more competitive due to its more efficient root finder, but still is slower than *crw* by about 20%. There is some evidence that *gtreeql* becomes more competitive as n grows, when the relative cost of the leaf solves becomes smaller. For an even larger problem ($n = 1024$) the ratio of *crw* time to *gtreeql* time on Problem 2 is 0.52. Recall that even when eigenvectors of the full problem are not required, *gtreeql* must still compute

Table 5: Sequential performance (eigenvalues only): time and speedup over *crw*.

	n	*crw*	*gtreeql*		*gtreps1*		*gtreps2*	
Problem 1	32	0.15	1.02	0.15	1.09	0.14	0.22	0.68
	64	0.57	2.29	0.25	4.10	0.14	0.78	0.73
	128	2.24	6.56	0.34	15.61	0.14	2.82	0.79
	256	8.84	19.65	0.45	60.29	0.15	10.96	0.81
	512	34.71	64.04	0.54	240.38	0.14	42.52	0.82
Problem 2	32	0.14	1.02	0.14	1.10	0.13	0.24	0.58
	64	0.56	2.87	0.20	4.08	0.14	0.85	0.66
	128	2.17	8.52	0.25	15.50	0.14	3.14	0.69
	256	8.64	26.84	0.32	59.96	0.14	12.02	0.72
	512	34.38	84.64	0.41	235.59	0.15	47.14	0.73

Table 6: Performance of the methods on a set of random matrices: residual and orthogonality.

	Residual			Orthogonality		
Method	Min	Max	Ave	Min	Max	Ave
crw	2.6×10^{-14}	5.5×10^{-14}	3.7×10^{-14}	2.3×10^{-14}	3.6×10^{-14}	3.0×10^{-14}
gtreeql	3.7×10^{-14}	1.9×10^{-11}	6.7×10^{-13}	2.3×10^{-14}	1.5×10^{-11}	5.4×10^{-13}
gtreps1	4.0×10^{-14}	7.5×10^{-13}	2.6×10^{-13}	7.8×10^{-16}	1.8×10^{-15}	1.1×10^{-15}
gtreps2	1.8×10^{-14}	5.9×10^{-13}	1.1×10^{-13}	8.9×10^{-16}	1.6×10^{-15}	1.1×10^{-15}

eigenvectors for the leaf problems, since the first and last components of each eigenvector are needed in order to perform the update step. Hence, it is likely that splitting the problem down to smaller leaf problems (i.e., using more divide-and-conquer levels) would improve the performance of *gtreeql* for the eigenvalue-only case. Section 4.7 looks briefly at this possibility.

4.3 Performance on a set of random matrices

In order to compare the performance of the methods considered here on a slightly wider set of problems, we tested each method on a set of matrix pencils $T - \lambda S$, where t_{ii} and $t_{i-1,i}$ are randomly distributed in $[-1, 1]$, $S = \text{tridag}(1/4, 1, 1/4)$, and $n = 256$. Tables 6 and 7 give the results. Table 7 also lists the number of problems that were solved by each method. Of the 100 cases tried, *gtreeql* got very poor answers on four of them due to a failure to converge in the root-finder. These are not reflected in the tables. We are working on a more robust strategy to handle cases such as these (see [3]). Notice that the accuracy for the current version of *gtreeql* is not as good as for the other methods, but that the time is substantially better. There is considerable deflation with these problems, and that accounts for the significant savings in time with *gtreeql*. The parallel performance of the methods on this set of problems is the same as on the other two test problems.

4.4 Parallel performance

Table 7: Performance of the methods on a set of random matrices: time.

Method	nprobs	Time Min	Time Max	Time Ave
crw	25	420.7	441.2	432.5
gtreeql	96	25.5	33.2	29.5
gtreps1	100	227.2	228.4	227.8
gtreps2	100	179.2	180.2	179.8

Table 8: Parallel performance: time, speedup over one processor time, and parallel efficiency.

		gtreeql			*gtreps1*			*gtreps2*		
	P	Time	Spd	Eff	Time	Spd	Eff	Time	Spd	Eff
Problem 1	1	390.4	–	–	1586.2	–	–	1374.1	–	–
	2	197.8	1.97	0.99	790.7	2.01	1.00	692.3	1.98	0.99
	4	101.5	3.85	0.96	394.4	4.02	1.01	345.9	3.97	0.99
	8	54.0	7.23	0.90	199.4	7.95	0.99	177.2	7.76	0.97
	16	32.0	12.22	0.76	101.4	15.64	0.98	89.7	15.32	0.96
Problem 2	1	509.4	–	–	1569.3	–	–	1375.2	–	–
	2	259.0	1.97	0.98	788.3	1.99	1.00	687.0	2.00	1.00
	4	131.6	3.87	0.97	395.5	3.97	0.99	347.0	3.96	0.99
	8	66.9	7.61	0.95	200.3	7.84	0.98	177.1	7.77	0.97
	16	36.6	13.93	0.87	102.7	15.28	0.95	90.8	15.15	0.95

Table 9: Parallel performance (eigenvalues only): time, speedup over one processor time, and parallel efficiency.

		gtreeql			*gtreps1*			*gtreps2*		
	P	Time	Spd	Eff	Time	Spd	Eff	Time	Spd	Eff
Problem 1	1	64.0	–	–	240.4	–	–	42.5	–	–
	2	34.3	1.87	0.93	119.4	2.01	1.01	21.3	2.00	1.00
	4	18.7	3.42	0.86	59.4	4.04	1.01	11.1	3.83	0.96
	8	10.7	6.00	0.75	30.5	7.88	0.99	6.5	6.53	0.82
	16	7.6	8.44	0.53	15.5	15.50	0.97	3.6	11.81	0.74
Problem 2	1	84.6	–	–	235.6	–	–	47.1	–	–
	2	44.7	1.89	0.95	117.7	2.00	1.00	24.9	1.89	0.95
	4	23.8	3.55	0.89	61.2	3.85	0.96	13.5	3.49	0.87
	8	11.8	7.19	0.90	30.9	7.63	0.95	7.4	6.36	0.80
	16	6.1	13.88	0.87	16.4	14.39	0.90	4.5	10.47	0.65

In Tables 8 and 9 we summarize the parallel performance of *gtreeql*, *gtreps1*, and *gtreps2*. Time in seconds, parallel speedup, and parallel efficiency (Spd/P) are given for P = 1, 2, 4, 8, and 16 processors. Note that speedup is measured relative to performance of the same code run on a single processor. If we compared timings with the best serial code (e.g., *crw* in the eigenvalue-only case), the speedups would be very different. We report speedup in this way so that the sequential computational advantages of one algorithm over against another are isolated from parallel performance issues. Since we give the actual time in all cases, the fastest overall method is still easily seen from the data.

It can be seen that the parallel efficiencies for *gtreeql* are relatively worse on Problem 1 than on Problem 2. This is primarily due to the greater number of deflations occurring in the first problem, leading to an unbalanced load and reducing the proportion of work that parallelizes well. If deflation occurs in an extremely nonuniform manner, the computational costs of the update steps at a given level may vary by as much as a factor of two. For example, on Problem 1, with $n = 256$, six of the leaf nodes have 16 deflations each while the other two nodes have no deflations.

In comparing Tables 8 and 9, it is also clear that parallel efficiencies are better when the eigenvector calculations are done. The reason is simple: a greater percentage of the work parallelizes well when the eigenvector calculations are required (orthogonalization being the dominant example). Stated another way, the proportion of the computation done serially is generally not significant when the eigenvector calculations are required. For example, when multiple eigenvalues are encountered in *gtreeql*, each deflation requires $O(n)$ work to carry out a Givens rotation. This is not currently parallelized, and is not a performance bottleneck—*unless* we have $O(n)$ deflations, as in Problem 1, and the remaining computation is also $O(n^2)$, as it is in the eigenvalue-only case. Thus, we see parallel efficiencies deteriorating very rapidly for *gtreeql* on Problem 1 when no eigenvectors are computed. It is possible to parallelize the deflation computation to some extent, but it would not be nearly as straightforward as the parallelization we have done thus far.

Overall, *gtreeql* is fastest if all the eigenvectors are required. However, as we have seen in §4.2 this advantage is entirely due to its sequential efficiency. In fact, the two *gtreps* methods parallelize more efficiently. This suggests a scalability advantage for *gtreps*, in that more processors can be used without seriously degrading the parallel efficiency. If only eigenvalues are desired, *gtreps2* is fastest (when it works); *gtreeql* is faster than *gtreps1*, but again we see better parallel efficiencies for the latter.

4.5 Orthogonalization in *gtreps*

As we have seen, *gtreps* orthogonalizes each eigenvector against the other eigenvectors in the same group. Recall that a group is defined in terms of eigenvalue separations of at least $\varepsilon = \delta\|T\|\mathrm{cond}_2(S)$, where $\delta = 10^{-3}$ by default. With this default choice for ε, for our two test problems and $n = 512$, all eigenvalues are in a single group, so that all n eigenvectors are orthogonalized against one another. For smaller n, a large enough gap occurs between a few of the eigenvalues that some smaller groups are defined. The result is significant savings in the cost of orthogonalization. However, for the purposes of the data reported in §§4.1–4.4 we forced all eigenvectors to be orthogonalized against each other (essentially setting $\delta = \infty$), so that the algorithm behaves in as uniform a way as possible as n is increased.

Obviously, the performance of *gtreps* can be quite sensitive to the grouping of eigenvalues. To get a feel for the effect of changing the heuristic used to define the groups, we re-ran *gtreps2* on Problem 1 with $n = 256$ and for various choices of δ (see Table 10). The last column in Table 10 lists the number of groups that were defined for a given choice of δ. There is a very clear effect

Table 10: Sequential performance of *gtreps2* on Problem 1 ($n = 256$) as grouping criterion varies.

δ	Residual	Orthogonality	Time	Ngroups
10^{-2}	3.0×10^{-14}	1.7×10^{-15}	177.9	1
10^{-3}	1.5×10^{-13}	1.5×10^{-13}	28.3	183
10^{-4}	1.5×10^{-13}	3.2×10^{-12}	17.3	249
10^{-5}	1.5×10^{-13}	5.7×10^{-11}	17.2	256

in both time and orthogonality as δ is reduced. For this problem, where the entire spectrum is contained in a relatively small interval, it appears that full orthogonalization is required to achieve eigenvectors that are orthogonal to near machine precision. For a problem which has well separated eigenvalues on the other hand, *gtreps* could take advantage of this and realize significant savings in orthogonalization.

4.6 Granularity in *gtreeql*

As described in §2.3, there are three main parallel tasks in our implementation of *gtreeql*: independent *crw* solves of the leaf problems, rank-one update computations, and eigenpair calculations (root-solves, etc.) within an update. The latter two need a little explanation. Since the goal is to balance useful work as much as possible, if there are enough rank-one updates on a given level of the control flow tree for each processor to have at least one, then no further parallelizing of the root-solves within an update is necessary. However, if there are not enough update tasks to go around, the root-solves are distributed among the processors. In order to keep the granularity of these root-solving tasks high, a single task consists of as many root-solves as possible. For example, with 16 processors we divide the root-solves needed for the largest update step into 32 groups or "bins"; at the next level in the tree each of the two updates divides its root-solves into 16 bins of more or less equal size; and so on. Notice that if deflation occurs non-uniformly, it is possible that the bins defined by one update node are somewhat smaller than those defined by another. In practice this does not appear to be a serious problem. It is extremely important, however, to define a sufficient total number of bins, especially at the root node. For example, on Problem 2 with $n = 512$, *gtreeql* spends nearly 65% of its total (sequential) time executing the last update. If the number of total bins generated at that point is set to 20 instead of 32, then the time taken by *gtreeql* on 16 processors increases from 36.6 to 53.1 seconds. Setting the total number of bins equal to the number of processors might seem like a natural choice, but we find that having approximately twice as many bins as processors is more efficient since it reduces the potential load imbalance due to bins of different sizes.

4.7 Divide-and-conquer depth in *gtreeql*

It is very difficult to determine an optimal size for the leaf problems solved by *gtreeql*. If such an optimum value exists, it must depend on the machine (e.g., number of processors, memory configuration, communication and synchronization costs, etc.), the problem (e.g., amount of deflation), and the desired solution (e.g., whether eigenvectors are required). Our experience is that for the class of machine we are considering here, choosing *nsmall* so that the number of leaves is approximately the same as the number of processors is a reasonable choice. (This strategy lies behind

Table 11: Time for *gtreeql* on Problem 2 for various values of *nsmall*.

		nsmall			
	n	8	16	32	64
Full computation	64	2.97	3.04	3.72	6.88
	128	15.48	15.61	16.91	23.29
	256	86.06	86.06	89.17	101.94
	512	502.96	501.78	509.37	533.53
Eigenvalues only	64	1.75	1.84	2.87	6.88
	128	6.36	6.61	8.52	16.76
	256	22.60	23.08	26.84	43.59
	512	76.06	77.01	84.64	117.83

the choice of *nsmall* = 32 in the data reported above, since our target configuration is $n = 512$ and 16 processors.) The data in Table 11 illustrate the sequential behavior of *gtreeql* on Problem 2 as *nsmall* varies. One can see that there is some advantage to smaller values of *nsmall*, especially for small n and for the eigenvalue only case. This is to be expected, since reducing *nsmall* tends to increase the advantage in computational complexity due to the divide-and-conquer approach. However, for larger n, and for multiple processors, the differences are really not that significant.

5 Summary and Conclusions

In this paper we have described generalizations of two well known algorithms for the symmetric tridiagonal eigenvalue problem. The generalized algorithms, *gtreeql* and *gtreps*, compute eigenpairs of a generalized symmetric tridiagonal eigenvalue problem $T\mathbf{x} = \lambda S\mathbf{x}$. We have compared the performance of these two algorithms on two test problems of varying sizes, and for varying number of processors on a shared memory multiprocessor. The results of the numerical experiments can be summarized as follows:

Accuracy. Both algorithms achieve accuracy near machine precision on the problems tested. However, there is some loss of orthogonality with increasing problem size for our present implementation of *gtreeql*; this issue is discussed in detail, with suggested improvements, in [3]. The *gtreps* algorithms require an expensive orthogonalization step unless eigenvalues are grouped and separated. Note that the spectrum for both test problems lies in a relatively small interval ($[0, 2]$ for Problem 1 and $[-.12, 2.49]$ for Problem 2), and that the eigenvalues have a quasiuniform distribution in the sense that they do not appear in clusters relative to the grouping criterion of *gtreps*. Problems with qualitatively different eigenvalue distributions may yield different results. In fact, a significant difference between divide-and-conquer methods and bisection methods seems to be that the latter can take better advantage of clustering by avoiding unnecessary orthogonalization. With divide-and-conquer, each subproblem must be solved as accurately as possible (with full orthogonalization) because the algorithm cannot know what the eigenvalue distribution of the global problem is going to be. Finally, note that *gtreps2* can suffer from overflow problems for large n.

Sequential performance. If eigenvectors are needed, *gtreeql* is the most efficient algorithm. If the orthogonalization costs can be reduced, *gtreps* is competitive as well. For the eigenvalue only case *crw* is the fastest, although *gtreps2* and *gtreeql* become competitive as problem size grows.

Parallel performance. The *gtreps* algorithms have better parallel efficiencies than *gtreeql* (comparing time for the same code on 1 and P processors). The parallel performance of *gtreeql* is sensitive to the depth of the divide-and-conquer tree, the granularity of root-finding tasks, and the pattern of deflation. Because of its advantages as a sequential algorithm, *gtreeql* is still superior in most of the cases tested. However, for fixed problem size, our data suggests that adding processors will benefit *gtreps* relatively more than *gtreeql.* Further analysis and experiments are needed to predict relative performance in terms of scaled speedup (i.e., problem size growing with the number of processors).

A goal of this research is to develop "rules of thumb" which help decide which approach is best—as a function of problem size, number of eigenpairs needed, distribution of spectrum, and machine characteristics. Toward that end, we plan to consider these algorithms on problems with qualitatively different eigenvalue distributions. Other areas for future work include a parallel version of Crawford's algorithm, and implementations on a distributed memory architecture.

Acknowledgements

We are indebted to Jack Dongarra and Danny Sorensen for providing us with the *treeql* code, and to Ahmed Sameh for providing us with the *treps* code. We also acknowledge the support of the computational facilities at Argonne National Laboratory.

References

[1] J. L. Barlow. Error analysis of update methods for the symmetric eigenvalue problem. Technical Report CS-91-23, Department of Computer Science, The Pennsylvania State University, August 1991.

[2] C. Beattie and D. W. Fox. Schur complements and the Weinstein-Aronszajn theory for modified matrix eigenvalue problems. *Lin. Alg. & Appl.*, 108:37–61, 1988.

[3] C. Beattie and C. J. Ribbens. A divide and conquer algorithm for the generalized symmetric tridiagonal eigenvalue problem. In preparation, 1992.

[4] A. E. Berger, J. M. Solomon, and M. Ciment. Higher order accurate tridiagonal difference methods for diffusion convection equations. In R. Vichnevetsky and R.S Stepleman, editors, *Advances in Computer Methods for Partial Differential Equations III*, pages 322–330. IMACS, 1978.

[5] R. F. Boisvert. Families of high order accurate discretizations of some elliptic problems. *SIAM J. Sci. Statist. Comput.*, 2:268–284, 1981.

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

[7] L. Collatz. *The Numerical Treatment of Differential Equations.* Springer, Berlin, 1960.

[8] C. R. Crawford. Reduction of a band-symmetric generalized eigenvalue problem. *Comm. ACM*, 16:41–44, 1973.

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

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

[11] J. J. Dongarra and D. C. Sorensen. SCHEDULE: an aid to programming explicitly parallel algorithms in FORTRAN. In A. Wouk, editor, *Parallel Processing and Medium-Scale Multiprocessors*, pages 177–191. Society for Industrial and Applied Mathematics, Philadelphia, 1989.

[12] K. Gates. Using inverse iteration to improve the divide and conquer algorithm. Technical Report 159, Departement Informatik, ETH, Zurich, 1991.

[13] G. H. Golub. Some modified matrix eigenvalue problems. *SIAM Rev.*, 15:318–344, 1973.

[14] B. Grieves and D. Dunn. Symmetric matrix methods for Schrödinger eigenvectors. *J. Phys. A: Math Gen.*, 23:5479–5491, 1990.

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

[16] E. R. Jessup. Parallel solution of the symmetric tridiagonal eigenvalue problem. Technical Report YALEU/DCS/RR-728, Yale University, 1989.

[17] L. Kaufman. Banded eigenvalue solvers on vector machines. *ACM Trans. Math. Softw.*, 10:73–86, 1984.

[18] A. S. Krishnakumar and M. Morf. Eigenvalues of a symmetric tridiagonal matrix: a divide-and-conquer approach. *Numer. Math.*, 48:349–368, 1986.

[19] T.-Y. Li and N. H. Rhee. Homotopy algorithm for symmetric eigenvalue problems. *Numer. Math.*, 55:265–280, 1989.

[20] T.-Y. Li, H. Zhang, and X.-H. Sun. Parallel homotopy algorithm for the symmetric tridiagonal eigenvalue problem. *SIAM J. Sci. Statist. Comput.*, 12:469–487, 1991.

[21] S.-S. Lo, B. Philippe, and A. Sameh. A multiprocessor algorithm for the symmetric tridiagonal eigenvalue problem. *SIAM J. Sci. Statist. Comput.*, 8:s155–s165, 1987.

[22] S. C. Ma, M. L. Patrick, and D. B. Szyld. A parallel, hybrid algorithm for the generalized eigenproblem. In G. Rodrigue, editor, *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 82–86, Philadelphia, 1989. Society for Industrial and Applied Mathematics.

[23] R. Natarajan. A parallel algorithm for the generalized symmetric eigenvalue problem on a hybrid multiprocessor. *Parallel Computing*, 14:129–150, 1990.

[24] R. D. Pantazis and D. B. Szyld. A multiprocessor method for the solution of the generalized eigenvalue problem on an interval. In J. Dongarra, P. Messina, D. C. Sorensen, and R. G. Voigt, editors, *Proceedings of the Fourth SIAM Conference on Parallel Processing for Scientific Computing*, pages 36–41, Philadelphia, 1990. Society for Industrial and Applied Mathematics.

[25] W. Shougen and Z. Shuqin. An algorithm for $Ax = \lambda Bx$ with symmetric and positive-definite A and B. *SIAM J. Matrix Anal. Appl.*, 12:654–660, 1991.

[26] D. C. Sorensen and P. T. P. Tang. On the orthogonality of eigenvectors computed by divide-and-conquer techniques. Technical Report MCS-P152-0490, Argonne National Laboratory, May 1990.

[27] A. van der Sluis. Condition numbers and equilibration of matrices. *Numer. Math.*, 14:14–23, 1969.

[28] J. S. Weston, M. Clint, and C. W. Bleakney. The parallel computation of eigenvalues and eigenvectors of large hermitian matrices using the AMT DAP 510. *Concurrency: Pract. & Exp.*, 3:179–185, 1991.