SIAM J. Sci. Comput.
Vol. 27, No. 1, pp. 43–66
© 2005 Society for Industrial and Applied Mathematics

A PARALLEL EIGENSOLVER FOR DENSE SYMMETRIC MATRICES BASED ON MULTIPLE RELATIVELY ROBUST REPRESENTATIONS[*]

PAOLO BIENTINESI[†], INDERJIT S. DHILLON[†], AND ROBERT A. VAN DE GEIJN[†]

Abstract. We present a new parallel algorithm for the dense symmetric eigenvalue/eigenvector problem that is based upon the tridiagonal eigensolver, Algorithm MR^3, recently developed by Dhillon and Parlett. Algorithm MR^3 has a complexity of $O(n^2)$ operations for computing all eigenvalues and eigenvectors of a symmetric tridiagonal problem. Moreover the algorithm requires only $O(n)$ extra workspace and can be adapted to compute any subset of k eigenpairs in $O(nk)$ time. In contrast, all earlier stable parallel algorithms for the tridiagonal eigenproblem require $O(n^3)$ operations in the worst case, while some implementations, such as divide and conquer, have an extra $O(n^2)$ memory requirement. The proposed parallel algorithm balances the workload equally among the processors by traversing a matrix-dependent *representation tree* which captures the sequence of computations performed by Algorithm MR^3. The resulting implementation allows problems of very large size to be solved efficiently—the largest dense eigenproblem solved in-core on a 256 processor machine with 2 GBytes of memory per processor is for a matrix of size 128,000 × 128,000, which required about 8 hours of CPU time. We present comparisons with other eigensolvers and results on matrices that arise in the applications of computational quantum chemistry and finite element modeling of automobile bodies.

Key words. parallel computing, symmetric matrix, eigenvalues, eigenvectors, relatively robust representations

AMS subject classifications. 65F15, 65Y05, 68W10

DOI. 10.1137/030601107

1. Introduction.

The symmetric eigenvalue problem is ubiquitous in computational sciences; problems of ever-growing size arise in applications as varied as computational quantum chemistry, finite element modeling, and pattern recognition. In many of these applications, both time and space are limiting factors for solving the problem, and hence efficient parallel algorithms and implementations are needed. The best approach for computing all the eigenpairs (eigenvalues and eigenvectors) of a dense symmetric matrix involves three phases: (1) *reduction*—reduce the given symmetric matrix A to tridiagonal form T; (2) *solution of tridiagonal eigenproblem*—compute all the eigenpairs of T; (3) *backtransformation*—map T's eigenvectors into those of A. For an $n \times n$ matrix, the reduction and backtransformation phases require $O(n^3)$ arithmetic operations each. Until recently, all algorithms for the tridiagonal eigenproblem too had cubic complexity in the worst case; these include the remarkable QR algorithm [26, 27, 36], inverse iteration [42], and the divide and conquer method [10].

In fact, the tridiagonal problem can be the computational bottleneck for large problems taking nearly 70–80% of the total time to solve the entire dense problem. For example, on a 2.4 GHz Intel Pentium 4 processor the tridiagonal reduction and backtransformation of a 2000 × 2000 dense matrix takes about 32 seconds, while

*Received by the editors October 6, 2003; accepted for publication (in revised form) January 11, 2005; published electronically August 17, 2005. This research was supported in part by NSF CAREER Award ACI-0093404 (ISD) and NSF Award ACI-0203685.
http://www.siam.org/journals/sisc/27-1/60110.html

†Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712 (pauldj@cs.utexas.edu, inderjit@cs.utexas.edu, rvdg@cs.utexas.edu).

LAPACK's bisection and inverse iteration software takes 106 seconds to compute all the eigenpairs of the tridiagonal. The timings for a 4000×4000 matrix clearly show the $O(n^3)$ behavior: 290 seconds for tridiagonal reduction and backtransformation, and 821 seconds for bisection and inverse iteration to solve the tridiagonal eigenproblem. Timings for the tridiagonal QR algorithm are 86 seconds for $n = 2000$ and 1099 seconds for $n = 4000$. More detailed timing results are given in section 4.

Recently, Dhillon and Parlett proposed Algorithm MRRR or MR^3 (multiple relatively robust representations) [13, 18, 17], which gives the first stable $O(nk)$ algorithm to compute k eigenvalues and eigenvectors of a symmetric tridiagonal matrix. In this paper we present a parallel algorithm based on Algorithm MR^3 for computing any subset of eigenpairs of a dense symmetric matrix; this yields the first parallel implementation of Algorithm MR^3. We refer to the parallel algorithm as PMR^3 (parallel MR^3). As a consequence, the time spent by the proposed algorithm on the tridiagonal eigenproblem is negligible compared to the time spent on reduction and backtransformation. For example, to compute all the eigenpairs of a 15,000 $\times$ 15,000 matrix on 16 processors the new algorithm requires 546 seconds for reduction, 22.2 seconds for the tridiagonal solution, and 160 seconds for backtransformation. In comparison, the corresponding timings for existing implementations for the tridiagonal eigensolution are 2054 seconds for the QR algorithm and 92.4 seconds for the divide and conquer method. For a 32,000 $\times$ 32,000 matrix the timings for PMR^3 on 16 processors are 4876 seconds for the reduction, 118 seconds for the tridiagonal solution, and 1388 seconds for backtransformation. These timings clearly contrast the $O(n^2)$ complexity of Algorithm MR^3 as opposed to the $O(n^3)$ reduction and backtransformation phases.

Moreover, some of the earlier algorithms have extra memory requirements: the ScaLAPACK divide and conquer code (PDSTEVD) requires extra $O(n^2)$ storage, while the inverse iteration code (PDSTEIN) can lead to a memory imbalance on the processors depending upon the eigenvalue distribution. Thus neither PDSTEVD nor PDSTEIN can be used to solve the above-mentioned 32,000 $\times$ 32,000 eigenproblem on 16 processors. In contrast, our parallel algorithm requires only workspace that is linear in n and the memory needed to store the eigenvectors of the tridiagonal problem is evenly divided among processors, thus allowing us to efficiently solve problems of very large size. The largest dense problem we have solved "in-core" on a 256 processor machine with 2 GBytes of memory per processor is a matrix of size 128,000 $\times$ 128,000, which required about 8 hours of computation time. Section 4 contains further timing results for the parallel implementations.

The rest of the paper is organized as follows. Section 2 reviews previous work on algorithms for the dense symmetric eigenvalue problem. In section 3, we present the proposed parallel Algorithm PMR^3, which uses multiple relatively robust representations for the tridiagonal problem. Section 4 presents detailed timing results comparing Algorithm PMR^3 with existing software. These include results on matrices that arise in the real-life applications of computational quantum chemistry and finite element modeling of automobile bodies.

A word on the notation used throughout the paper. T indicates a symmetric tridiagonal matrix, n represents the size of a matrix, eigenvalues are denoted by λ, and eigenvectors are denoted by $\boldsymbol{v}$. Computed quantities will often be denoted by "hatted" symbols, for example, $\hat{\lambda}$ and $\hat{\boldsymbol{v}}$. The number of processors in a parallel computation is p, while the ith processor is denoted by p_i.

2. Related work.

As mentioned earlier, most algorithms for the dense symmetric eigenvalue problem proceed in three phases. The first and third phases, Householder reduction and backtransformation, are fairly standard and described in section 3.2. The second stage, tridiagonal eigensolution, has led to a variety of interesting algorithms; we now give a quick overview of existing methods, emphasizing their parallel versions.

The QR algorithm, independently invented by Francis [26, 27] and Kublanovskaya [36], is an iteration that produces a sequence of similar matrices that converges to diagonal form. When the starting matrix is symmetric and tridiagonal, each iterate produced by the QR algorithm is also symmetric and tridiagonal. Convergence to diagonal form is rapid (ultimately cubic) with a suitable choice of shifts [38]. A fast square-root-free version of QR developed by Pal, Walker, and Kahan (PWK) is useful if only eigenvalues are desired [38]. Another attractive alternative, in the latter case, is to use the differential quotient-difference algorithm (dqds) that is based on the related LR iteration [24]. In practice 2–3 iterations, on average, are needed per eigenvalue in the QR algorithm where each iteration is composed of at most $n-1$ Givens rotations. Thus all eigenvalues can be computed at a cost of $O(n^2)$ operations, while the accumulation of Givens rotations required for computing orthogonal eigenvectors results in $O(n^3)$ operations (in practice, $6n^3$ to $9n^3$ operations are observed).

The inherent sequential nature of the QR algorithm makes the eigenvalue computation hard to parallelize. However, when eigenvectors are needed, an effective parallel algorithm that yields good speedups can be obtained as follows. First, the Householder reflections computed during the reduction are accumulated in approximately $\frac{4}{3}n^3$ operations to form a matrix Z, which is then evenly partitioned among the p processors so that each processor owns approximately n/p rows of Z. The tridiagonal matrix is duplicated on all processors and the $O(n^2)$ eigenvalue computation is redundantly performed on all processors, while the Givens rotations are directly applied on each processor to its part of Z. This accumulation achieves perfect speedup since all processors can simultaneously update their portion of Z without requiring any communication, thus leading to an overall parallel complexity of $O(n^3/p)$ operations. A faster algorithm (up to a factor of 2) can be obtained by using perfect shifts and inner deflations [16].

A major drawback of the QR algorithm is that it is hard to adapt to the case when only a subset of eigenvalues and eigenvectors is desired at a proportionately reduced operation count. Thus a commonly used parallel solution is to invoke the bisection algorithm followed by inverse iteration [32]. The bisection algorithm was first proposed by Givens in 1954 and allows the computation of k eigenvalues of a symmetric tridiagonal T in $O(kn)$ operations [28]. Once accurate eigenvalues are known, the method of inverse iteration may be used to compute the corresponding eigenvectors [42]. However, inverse iteration can guarantee only small residual norms. It cannot ensure orthogonality of the computed vectors when eigenvalues are close. A commonly used "remedy" is to orthogonalize each approximate eigenvector, using the modified Gram–Schmidt method, against previously computed eigenvectors of "nearby" eigenvalues—the LAPACK and EISPACK implementations orthogonalize when eigenvalues are closer than $10^{-3}\|T\|$. Unfortunately even this conservative strategy can fail to give accurate answers in certain situations [14]. The amount of work required by inverse iteration to compute all the eigenvectors of a symmetric tridiagonal matrix depends strongly upon the distribution of eigenvalues (unlike the QR algorithm, which always requires $O(n^3)$ operations). If eigenvalues are well separated (gaps greater than $10^{-3}\|T\|$), then $O(n^2)$ operations are sufficient. However, when eigenvalues are clustered, current implementations of inverse iteration can take up to $10n^3$ operations due to orthogonalization [34]. Unfortunately the latter situation is the norm rather than the exception for large matrices since even uniform eigenvalue spacings when n exceeds 1000 can lead to eigenvalue gaps smaller than $10^{-3}\|T\|$.

When eigenvalues are well separated, both bisection and inverse iteration can be effectively parallelized leading to a complexity of $O(n^2/p)$ operations. However, as remarked above, the common situation for large matrices is that inverse iteration requires $O(n^3)$ operations; see section 4.3 for some timings. Thus, parallel inverse iteration requires $O(n^3/p)$ operations in these situations. Moreover, considerable communication is required when Gram–Schmidt orthogonalization is done across processor boundaries. Indeed, to avoid communication, the current inverse iteration implementation in ScaLAPACK (PDSTEIN) computes all the eigenvectors corresponding to a cluster of eigenvalues on a single processor, thus leading to a parallel complexity of $O(n^3)$ in the worst case and also an imbalance in the memory required on each processor [8, p. 48].

The bisection algorithm to find eigenvalues has linear convergence and can be quite slow. To speed up bisection, there have been many attempts to employ faster zero-finders such as the Rayleigh quotient iteration [38], Laguerre's method [37, 39], and the zeroin scheme [11, 9]. These zero-finders can speed up the computation of isolated eigenvalues by a considerable amount, but they seem to stumble when eigenvalues cluster. In all these cases, the corresponding eigenvectors still need to be computed by inverse iteration.

The divide and conquer method proposed by Cuppen in 1981 is a method specially suited for parallel computation [10, 20]; remarkably this algorithm also yields a faster sequential implementation than QR. The basic strategy of the divide and conquer algorithm is to express the tridiagonal matrix as a low-rank modification of a direct sum of two smaller tridiagonal matrices. This modification may be a rank-one update or may be obtained by crossing out a row and column of the tridiagonal. The entire eigenproblem can then be solved in terms of the eigenproblems of the smaller tridiagonal matrices, and this process can be repeated recursively. For several years after its inception, it was not known how to guarantee numerically orthogonality of the eigenvector approximations obtained by this approach. However, Gu and Eisenstat found a solution to this problem, leading to robust software based on their strategy [30].

The main reason for the unexpected success of divide and conquer methods on serial machines is *deflation*, which occurs when an eigenpair of a submatrix of T is an acceptable eigenpair of a larger submatrix. The greater the amount of deflation, which depends on the eigenvalue distribution and on the structure of the eigenvectors, the lesser the work required in these methods. For matrices with clustered eigenvalues, deflation can be extensive; however, in general, $O(n^3)$ operations are needed. The divide and conquer method is suited for parallelization since smaller subproblems can be solved independently on various processors. However, communication costs for combining subproblems are substantial, especially when combining the larger subproblems to get the solution to the full problem [43]. A major drawback of the divide and conquer algorithm is its extra $O(n^2)$ memory requirement—as we shall see later, this limits the largest problem that can be solved using this approach.

2.1. Other solution methods. The oldest method for solving the symmetric eigenproblem dates back to Jacobi [33]. Jacobi's method does not reduce the dense symmetric matrix to tridiagonal form, as most other methods do, but works on the dense matrix at all stages. It performs a sequence of plane rotations, each of which annihilates an off-diagonal element (which is filled in during later steps). There are a variety of Jacobi methods that differ solely in their strategies for choosing the next element to be annihilated. All good strategies tend to diminish the off-diagonal elements, and the resulting sequence of matrices converges to the diagonal matrix of eigenvalues.

Jacobi's method fell out of favor with the discovery of the QR algorithm. The primary reason is that, in practice, the cost of even the most efficient variants of the Jacobi iteration is an order of magnitude greater than that of the QR algorithm. Nonetheless, it has periodically enjoyed a resurrection since it can be efficiently parallelized, and theoretical results show it to be more accurate than the QR algorithm [12].

The symmetric invariant subspace decomposition algorithm (SYISDA) formulates the problem in a dramatically different way [5]. The idea is to scale and shift the spectrum of the given matrix so that its eigenvalues are mapped to the interval $[0, 1]$, with the mean eigenvalue being mapped to $\frac{1}{2}$. Letting B equal the transformed matrix, a polynomial p is applied to B with the property that $\lim_{i\to\infty} p^i([0, 1]) = \{0, 1\}$. By applying this polynomial to B in the iteration $C_0 = B$, $C_{i+1} = p(C_i)$ until convergence, all eigenvalues of C_{i+1} eventually become arbitrarily close to 0 or 1. The eigenvectors of C_{i+1} and A are related in such a way that allows the computation of two subspaces that can then be used to decouple matrix A into two subproblems, each of size roughly half that of A. The process then continues with each of the subproblems. The benefit of the SYISDA approach is that the computation can be cast in terms of matrix-matrix multiplication, which can attain near-peak performance on modern microprocessors and parallelizes easily [19, 31]. Unfortunately, this benefit is accompanied by an increase in the operation count, to the point where SYISDA is not considered to be competitive.

2.2. Parallel libraries. A great deal of effort has been spent in building efficient parallel symmetric eigensolvers for distributed systems. Specially designed software for this problem has been developed as part of a number of numerical libraries. Among these the best known are the Scalable Linear Algebra Package (ScaLAPACK) [21, 8], Parallel Eigensolver (PeIGS) [23], the Parallel Research on Invariant Subspace Methods (PRISM) project [5], and the Parallel Linear Algebra Package (PLAPACK) [44]. All of these packages attempt to achieve portability by embracing the Message-Passing Interface (MPI) and the Basic Linear Algebra Subprograms (BLAS) [19].

The ScaLAPACK project is an effort to parallelize the Linear Algebra Package (LAPACK) [1] to distributed memory architectures. It supports a number of different algorithms, as further discussed in the experimental section. PeIGS supports a large number of chemistry applications that give rise to large dense eigenvalue problems. It includes a parallel tridiagonal eigensolver that is based on a very early version of Algorithm MR^3; this preliminary version does limited Gram–Schmidt orthogonalization; see [15]. The PRISM project implements the SYISDA approach outlined in section 2.1. PLAPACK currently supports a parallel implementation of the QR algorithm as well as the algorithm that is the topic of this paper.

3. The proposed algorithm. We now present the proposed parallel algorithm. Section 3.1 describes how the tridiagonal eigenproblem can be solved using the method of multiple relatively robust representations (MR^3), while section 3.2 briefly describes the phases of Householder reduction and backtransformation.

3.1. Tridiagonal eigensolver using multiple relatively robust representations. Algorithm MR^3 was recently introduced by Dhillon and Parlett [13, 18, 17] for the task of computing k eigenvectors of a symmetric tridiagonal matrix T, and has a complexity of $O(nk)$ operations. The superior time complexity of the algorithm is achieved by avoiding Gram–Schmidt orthogonalization, which in turn is the result of high relative accuracy in intermediate computations.

3.1.1. The sequential algorithm. We provide the main ideas behind Algorithm MR^3; an in-depth technical description and justification of the algorithm can be found in [13, 18, 17]. There are three key ingredients that form the backbone of Algorithm MR^3:

1. *Relatively robust representations.* A relatively robust representation (RRR) is a representation that determines its eigenvalues and eigenvectors to high relative accuracy; i.e., small componentwise changes to individual entries of the representation lead to small relative changes in the eigenvalues and small changes in the eigenvectors (modulo relative gaps between eigenvalues; see (2) below). Unfortunately, the traditional representation of a tridiagonal by its diagonal and off-diagonal elements does not form an RRR; see [18, sect. 3] for an example. However, the bidiagonal factorization $T = LDL^t$ of a positive definite tridiagonal is an RRR, and in many cases an indefinite LDL^t also forms an RRR [18]. We now make precise the conditions needed for LDL^t to be an RRR: Write l_i for $L(i+1,i)$ and d_i for $D(i,i)$. Define the relative gap of $\hat{\lambda}$, where $\hat{\lambda}$ is closer to λ than to any other eigenvalue of LDL^t, to be

$$\text{relgap}(\hat{\lambda}) := \text{gap}(\hat{\lambda})/|\hat{\lambda}|,$$

where $\text{gap}(\hat{\lambda}) = \min\{|\nu - \hat{\lambda}| : \nu \neq \lambda,\ \nu \in \text{spectrum}(LDL^t)\}$. We say that $(\lambda, \boldsymbol{v})$ is determined to high relative accuracy by L and D if small relative changes, $l_i \to l_i(1+\eta_i)$, $d_i \to d_i(1+\delta_i)$, $|\eta_i| < \xi$, $|\delta_i| < \xi$, $\xi \ll 1$, cause changes $\delta\lambda$ and $\delta\boldsymbol{v}$ that satisfy

$$\frac{|\delta\lambda|}{|\lambda|} \leq K_1 n\xi, \quad \lambda \neq 0, \tag{1}$$

$$|\sin\angle(\boldsymbol{v}, \boldsymbol{v}+\delta\boldsymbol{v})| \leq \frac{K_2 n\xi}{\text{relgap}(\lambda)} \tag{2}$$

for modest constants K_1 and K_2, say, smaller than 100. We call such an LDL^t factorization an RRR for $(\lambda, \boldsymbol{v})$. The advantage of an RRR is that the eigenvalues and eigenvectors can be computed to high relative accuracy as governed by (1) and (2). For more details see [18].

2. *Computing the eigenvector of an isolated eigenvalue.* Once an accurate eigenvalue $\hat{\lambda}$ is known, its eigenvector may be computed by solving the equation $(LDL^t - \hat{\lambda}I)\boldsymbol{z} \approx 0$. However, it is not straightforward to solve this equation: the trick is to figure out what equation to ignore in this nearly singular system. Unable to find a solution to this problem, current implementations of inverse iteration in LAPACK and EISPACK solve $(LDL^t - \hat{\lambda}I)\boldsymbol{z}_{i+1} = \boldsymbol{z}_i$ and take $\boldsymbol{z}_0$ to be a random starting vector (this difficulty was known to Wilkinson [45, p. 318]). This problem was solved recently by using twisted factorizations that are obtained by gluing a top-down (LDL^t) and a bottom-up (UDU^t) factorization. The solution is presented in Algorithm Getvec below; see [18, 40, 25] for more details.

3. *Computing orthogonal eigenvectors for clusters using multiple RRRs.* By using an RRR and Algorithm Getvec for computing eigenvectors, it can be shown that the computed eigenvectors are numerically orthogonal when the eigenvalues have large relative gaps [18]. However, when eigenvalues have small relative gaps, the above approach is not adequate. For the case of small relative gaps, Algorithm MR^3 uses

```
ALGORITHM Getvec($L$,$D$, $\hat\lambda$).
Input:  $L$ is unit lower bidiagonal ($l_i$ denotes $L(i+1,i)$, $1 \le i \le n-1$), and $D$
        is diagonal ($d_i$ denotes $D(i,i)$, $1 \le i \le n$); $LDL^t$ is the input tridiagonal
        matrix assumed to be irreducible. $\hat\lambda$ is an approximate eigenvalue.
Output: $z$ is the computed eigenvector.
   I.   Factor $LDL^t - \hat\lambda I = L_+D_+L_+^t$ by the dstqds (differential stationary
        quotient-difference with shift) transform.
   II.  Factor $LDL^t - \hat\lambda I = U_-D_-U_-^t$ by the dqds (differential progressive quotient-
        difference with shift) transform.
   III. Compute $\gamma_k$ for $k = 1,\ldots,n$ by the formula $\gamma_k = s_k + \frac{d_k}{D_-(k+1)}p_{k+1}$ that
        involves the intermediate quantities $s_k$ and $p_{k+1}$ computed in the dstqds
        and dqds transforms (for details see [18, sect. 4.1]). Pick an $r$ such that
        $|\gamma_r| = \min_k |\gamma_k|$. Form the twisted factors with twist index $r$, $N_r$ and $\Delta_r$,
        which satisfy $N_r\Delta_r N_r^t = LDL^t - \hat\lambda I$.
   IV.  Form the approximate eigenvector $z$ by solving $N_r^t z = e_r$ ($e_r$ is the $r$th
        column of the identity matrix $I$), which is equivalent to solving $(LDL^t -
        \hat\lambda I)z = N_r\Delta_r N_r^t z = e_r\gamma_r$ since $N_r e_r = e_r$ and $\Delta_r e_r = \gamma_r e_r$:
            $z(r) = 1.$
            For $i = r-1,\ldots,1,\quad z(i) = \begin{cases} -L_+(i)z(i+1), & z(i+1) \ne 0, \\ -(d_{i+1}l_{i+1}/d_il_i)z(i+2), & \text{otherwise.} \end{cases}$
            For $j = r,\ldots,n-1,\quad z(j+1) = \begin{cases} -U_-(j)z(j), & z(j) \ne 0, \\ -(d_{j-1}l_{j-1}/d_jl_j)z(j-1), & \text{otherwise.} \end{cases}$
        Note that $d_il_i$ is the $(i,i+1)$ element of $LDL^t$.
   V.   Set $z \leftarrow z/\|z\|$.
```

FIG. 1. *Algorithm* Getvec *for computing the eigenvector of an isolated eigenvalue.*

multiple RRRs, i.e., multiple factorizations $L_cD_cL_c^t = LDL^t - \tau_c I$, where τ_c is close to a cluster. The shifts τ_c are chosen to "break" clusters, i.e., to make relative gaps bigger (note that relative gaps change upon shifting by τ_c). After forming the new representation $L_cD_cL_c^t$, the eigenvalues in the cluster are "refined" so that they have high relative accuracy with respect to $L_cD_cL_c^t$. Finally the eigenvectors of eigenvalues that become relatively well separated after shifting are computed by Algorithm Getvec using $L_cD_cL_c^t$; the process is iterated for eigenvalues that still have small relative gaps. Details are given in Algorithm MR^3 below. The tricky theoretical aspects that address the relative robustness of intermediate representations and whether the eigenvectors computed using different RRRs are numerically orthogonal may be found in [41] and [17]. It is important to note that orthogonality of the computed eigenvectors is achieved without Gram–Schmidt being used in any of the procedures.

We first present Algorithm Getvec in Figure 1. Getvec takes an LDL^t factorization and an approximate eigenvalue $\hat\lambda$ as input and computes the corresponding eigenvector by forming the appropriate twisted factorization $N_r\Delta_r N_r^t = LDL^t - \hat\lambda I$. The twist index r in step III of Figure 1 is chosen so that $|\gamma_r| = \min_k |\gamma_k|$ and is followed by solving $(LDL^t - \hat\lambda I)z = \gamma_r e_r$; thus r is the index of the equation that is ignored and provides a solution to Wilkinson's problem mentioned above [40]. The resulting eigenvector is accurate since differential transformations are used to compute the twisted factorization, and the eigenvector is computed solely by multiplications (no additions or subtractions) in step IV of the algorithm. We assume that LDL^t is an irreducible tridiagonal, i.e., all off-diagonals are nonzero. Details on twisted factorizations, differential quotient-difference transforms, and Algorithm Getvec may be found in [18].

Algorithm Getvec computes a single eigenvector of an RRR; it was shown in [18] that the computed eigenvector is highly accurate and so is numerically orthogonal to all other eigenvectors if the corresponding eigenvalue has a large relative gap. However, if Getvec is invoked when the corresponding eigenvalue is part of a cluster, the computed vector will, in general, not be orthogonal to other eigenvectors in the cluster. The difficulty is that, as seen by (2), the eigenvectors of eigenvalues with small relative gaps are highly sensitive to tiny changes in L and D.

To overcome this problem, Algorithm MR^3, given in Figure 2, uses multiple LDL^t factorizations—the basic idea is that there will be an LDL^t factorization for each cluster of eigenvalues. A new LDL^t factorization is formed per cluster in order to increase relative gaps within the cluster. Once an eigenvalue has a large relative gap, Algorithm Getvec is invoked to compute the corresponding eigenvector, as seen in step II of Figure 2. Otherwise we are in the presence of a cluster Γ_c of eigenvalues and a new representation $L_c D_c L_c^t = LDL^t - \tau_c I$ needs to be computed. The shift τ_c is chosen in such a way such that (a) the new representation is relatively robust for the eigenvalues in Γ_c and (b) at least one of the shifted eigenvalues in Γ_c is relatively well separated from the others. The process is iterated if other clusters are encountered. The inputs to MR^3 are an index set Γ_0 that specifies the desired eigenpairs, the symmetric tridiagonal matrix T given by its traditional representation of diagonal and off-diagonal elements, and a tolerance *tol* for relative gaps. Note that the computational path taken by MR^3 depends on the relative gaps between eigenvalues. We again emphasize that MR^3 does not need any Gram–Schmidt orthogonalization of the eigenvectors.

3.1.2. Representation trees. The sequence of computations in Algorithm MR^3 can be pictorially expressed by a *representation tree.* Such a tree contains information about how the eigenvalues are clustered (nodes of the tree) and what shifts are used to "break" a cluster (edges of the tree). A precise description of a representation tree can be found in [17]. Here we present a slightly simplified version of the tree, without specifying edge labels, which will facilitate the description of the parallel algorithm.

The root node of the representation tree is denoted by (L_0, D_0, Γ_0), where $L_0 D_0 L_0^t$ is the base representation obtained in step 1A of Algorithm MR^3; see Figure 2. An example representation tree is shown in Figure 3. Let Π_c be an internal node of the tree and let Π_p be its parent node. If Π_c is a nonleaf node, it will be denoted by (L_c, D_c, Γ_c), where the index set Γ_c is a proper subset of Γ_p, the index set of the parent node $\Pi_p = (L_p, D_p, \Gamma_p)$. Node Π_c indicates that $L_c D_c L_c^t$ is a representation that is computed by shifting, $L_p D_p L_p^t - \tau_c = L_c D_c L_c^t$, and will be used for computing the eigenvectors indexed by Γ_c. If Π_c is a leaf node instead, it will be denoted only by the singleton $\{c\}$, where $c \in \Gamma_p$. The singleton node $\{c\}$ signifies that the eigenvalue λ_c has a large relative gap with respect to the parent representation $L_p D_p L_p^t$, and its eigenvector will be computed by Algorithm Getvec.

Figure 3 gives an example of a representation tree for a matrix of size 11 for which all the eigenvectors are desired: the root contains the representation L_0, D_0 and the index set $\Gamma_0 = \{1, 2, \ldots, 11\}$. The algorithm begins by classifying the eigenvalues: in this example λ_1, λ_4, and λ_{11} are well separated, so in the tree they appear as singleton

```
ALGORITHM $\mathsf{MR}^3(T,\Gamma_0,tol)$.
  Input: $T$ is the given symmetric tridiagonal;
         $\Gamma_0$ is the index set of desired eigenpairs;
         $tol$ is the input tolerance for relative gaps, usually set to $10^{-3}$.
Output: $(\hat{\lambda}_j, \hat{\boldsymbol{v}}_j), j \in \Gamma_0$, are the computed eigenpairs.
1. Split $T$ into irreducible subblocks $T_1, T_2, \ldots, T_\ell$.
      For each subblock $T_i$, $i = 1, \ldots, \ell$, do:
         A. Choose $\mu_i$, and compute $L_0$ and $D_0$ such that $L_0 D_0 L_0^t = T_i + \mu_i I$ is a
              factorization that determines the desired eigenvalues and eigenvectors,
              $\lambda_j$ and $\boldsymbol{v}_j$, $j \in \Gamma_0$, to high relative accuracy. In general, the shift $\mu_i$ can
              be in the interior of $T_i$'s spectrum, but a safe choice is to make $T_i + \mu_i I$
              positive or negative definite.
         B. Compute the desired eigenvalues of $L_0 D_0 L_0^t$ to high relative accuracy
              by the dqds algorithm [24] or by bisection using a differential quotient-
              difference transform.
         C. Form a work queue $Q$, and initialize $Q = \{(L_0, D_0, \Gamma_0)\}$. Call
              $\mathsf{MR}^3\_\mathsf{Vec}(Q,tol)$.
      end for
Subroutine $\mathsf{MR}^3\_\mathsf{Vec}(Q,tol)$

While queue $Q$ is not empty:
I. Remove an element $(L, D, \Gamma)$ from the queue $Q$. Partition the computed eigen-
      values $\hat{\lambda}_j, j \in \Gamma$, into clusters $\Gamma_1, \ldots, \Gamma_h$ according to their relative gaps
      and the input tolerance $tol$. The eigenvalues are thus designated as iso-
      lated (cluster size equals 1) or clustered. More precisely, if $\mathsf{rgap}(\hat{\lambda}_j) :=$
      $\min_{i \neq j} |\hat{\lambda}_j - \hat{\lambda}_i|/|\hat{\lambda}_j| \geq tol$, then $\hat{\lambda}_j$ is isolated. On the other hand, all
      consecutive eigenvalues $\hat{\lambda}_{j-1}, \hat{\lambda}_j$ in a nontrivial cluster $\Gamma_c$ ($|\Gamma_c| > 1$) satisfy
      $|\hat{\lambda}_j - \hat{\lambda}_{j-1}|/|\hat{\lambda}_j| < tol$.
II. For each cluster $\Gamma_c$, $c = 1, \ldots, h$, perform the following steps.
         If $|\Gamma_c| = 1$ with eigenvalue $\hat{\lambda}_j$, i.e., $\Gamma_c = \{j\}$, then invoke Algo-
              rithm $\mathsf{Getvec}(L,D,\hat{\lambda}_j)$ to obtain the computed eigenvector $\hat{\boldsymbol{v}}_j$.
         else
              a. Pick $\tau_c$ near the cluster and compute $LDL^t - \tau_c I = L_c D_c L_c^t$ us-
                   ing the dstqds (differential form of stationary quotient-difference)
                   transform; see [18, sect. 4.1] for details.
              b. "Refine" the eigenvalues $\hat{\lambda} - \tau_c$ in the cluster so that they have
                   high relative accuracy with respect to the computed $L_c D_c L_c^t$. Set
                   $\hat{\lambda} \leftarrow (\hat{\lambda} - \tau_c)_{refined}$ for all eigenvalues in the cluster.
              c. Add $(L_c, D_c, \Gamma_c)$ to the queue $Q$.
         end if
      end for
end while
```

FIG. 2. *Algorithm* MR^3 *for computing orthogonal eigenvectors without using Gram–Schmidt orthogonalization.*

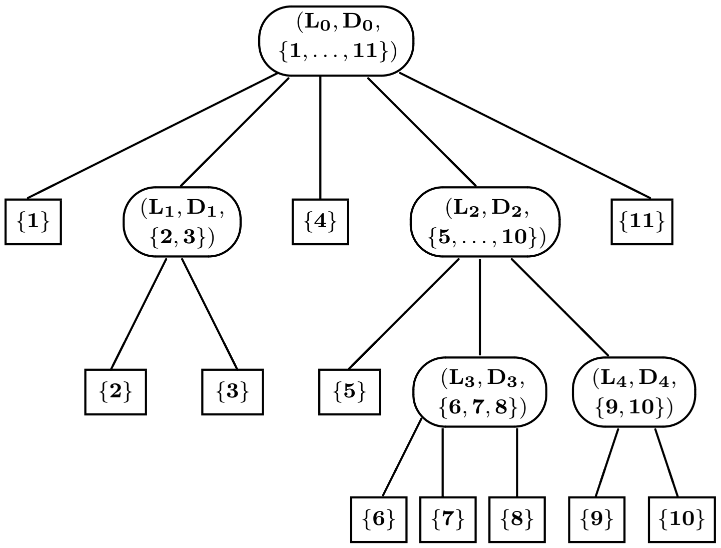
$(\mathbf{L_0}, \mathbf{D_0}, \{\mathbf{1}, \ldots, \mathbf{11}\})$
$\{\mathbf{1}\}$
$(\mathbf{L_1}, \mathbf{D_1}, \{\mathbf{2}, \mathbf{3}\})$
$\{\mathbf{4}\}$
$(\mathbf{L_2}, \mathbf{D_2}, \{\mathbf{5}, \ldots, \mathbf{10}\})$
$\{\mathbf{11}\}$
$\{\mathbf{2}\}$
$\{\mathbf{3}\}$
$\{\mathbf{5}\}$
$(\mathbf{L_3}, \mathbf{D_3}, \{\mathbf{6}, \mathbf{7}, \mathbf{8}\})$
$(\mathbf{L_4}, \mathbf{D_4}, \{\mathbf{9}, \mathbf{10}\})$
$\{\mathbf{6}\}$
$\{\mathbf{7}\}$
$\{\mathbf{8}\}$
$\{\mathbf{9}\}$
$\{\mathbf{10}\}$

FIG. 3. *An example representation tree for a matrix of size* 11.

leaves (their eigenvectors can be directly computed by the calls $\mathsf{Getvec}(L_0, D_0, \hat{\lambda}_i)$, $i = 1, 4$, and 11). The second and third eigenvalues violate the condition $|(\hat{\lambda}_3 - \hat{\lambda}_2)/\hat{\lambda}_3| \geq tol$, and therefore they form a cluster; a new representation $L_1 D_1 L_1^t = L_0 D_0 L_0^t - \tau_1 I$ has to be computed and the two eigenvalues have to be refined to have high relative accuracy with respect to L_1 and D_1. This is represented by the node $(L_1, D_1, \{2, 3\})$. Similarly the eigenvalues $\{\lambda_5, \ldots, \lambda_{10}\}$ are clustered: a new representation $L_2 D_2 L_2^t = L_0 D_0 L_0^t - \tau_2 I$ is computed as shown by the node $(L_2, D_2, \{5, \ldots, 10\})$. This illustrates the working of Algorithm MR^3 for all the nodes at depth 1 in the representation tree of Figure 3.

The computation proceeds by classifying the eigenvalues of the two internal nodes $(L_1, D_1, \{2, 3\})$ and $(L_2, D_2, \{5, \ldots, 10\})$. From the tree it can be deduced that the two eigenvalues λ_2 and λ_3 in the first internal node are now relatively well separated, while the second node is further fragmented into a relatively well separated eigenvalue λ_5 and two nodes with clusters: nodes $(L_3, D_3, \{6, 7, 8\})$ and $(L_4, D_4, \{9, 10\})$. Finally these two clusters are further fragmented to yield singletons, and these eigenvectors are computed by Algorithm Getvec. Note that, as seen by the description of Algorithm MR^3 in Figure 2, the representation tree is processed in a breadth-first fashion.

For most matrices, the depth of the representation tree is quite small; we give here a sketch of the representation tree for two matrices that arise in the finite element modeling of automobile bodies (see section 4.1 for more details). The tree for the matrix auto.13786 ($n =$ 13,786) has maximum depth 2; at depth 1 there are 12,937 singleton nodes, 403 clusters of size 2, 10 clusters of size 3, and 1 cluster of size 13. The tree for the matrix auto.12387 ($n =$ 12,387) also has maximum depth 2 even though it has many more internal nodes: it has 5776 singletons and 1991 nodes corresponding to clusters with sizes ranging from 2 to 31.

A further note about reducible matrices: the solution is computed by iterating over the subblocks, and thus the sequence of computations can be captured by a forest of trees (one for each subblock) rather than by a single tree.

3.1.3. The parallel algorithm. We now describe Algorithm PMR^3. The input to the algorithm is a tridiagonal matrix T, an index set Γ_0 of desired eigenpairs, a tolerance parameter *tol*, and the number of processors p that execute the algorithm. We target our algorithm to a distributed memory system, in which each processor has its own local main memory and communication is done by message-passing. We assume that the tridiagonal matrix is available on every processor before the algorithm is invoked.

We first discuss the parallelization strategy before describing the algorithm in detail. Let the size of the input index set Γ_0 be k; i.e., k eigenvalues and eigenvectors are to be computed. The total $O(kn)$ complexity of Algorithm MR^3 can be broken down into the work required at each node of the representation tree:

1. Each leaf node requires the computation of an eigenvector by Algorithm Getvec, which requires $O(n)$ operations (at most $2n$ divisions and $10n$ multiplications and additions).
2. Each internal node (L_c, D_c, Γ_c) requires (a) computation of the representation $L_c D_c L_c^t = L_i D_i L_i^t - \tau_c I$, and (b) refinement of the eigenvalues corresponding to the index set Γ_c so that they have high relative accuracy with respect to $L_c D_c L_c^t$. Computing the representation by the differential stationary quotient-difference transform requires $O(n)$ operations (n divisions, $4n$ multiplications and additions), while refinement of the eigenvalues can be done in $O(n|\Gamma_c|)$ operations using a combination of bisection and Rayleigh quotient iteration.

We aim for a parallel complexity of $O(\frac{nk}{p} + n)$ operations. Due to communication overheads, we will not attempt to parallelize $O(n)$ procedures, such as computing a single eigenvector or computing a new representation. Our strategy for the parallel algorithm will be to divide the leaf nodes equally among the processors, i.e., each processor will make approximately k/p calls to Algorithm Getvec. Thus each processor is assigned a set of eigenvectors that are to be computed locally.

However, before the leaf nodes can be processed the computation at the internal nodes needs to be performed. Each internal node (L_c, D_c, Γ_c) is associated with a subset of q processors, $q \leq p$, that are responsible for computing the eigenvectors in Γ_c. Since k eigenvectors are to be computed by a total of p processors, q approximately equals $\lceil p|\Gamma_c|/k \rceil$. Note that since $|\Gamma_c|$ is small in most practical applications (see comments towards the end of section 3.1.2), q is mostly small; in our examples, q is usually 1, sometimes 2, but rarely greater than 2. If q equals 1, the computation at each internal node is just done serially. If q is greater than 1, the parallel algorithm will process an internal node as follows: the representation $L_c D_c L_c^t$ is computed (redundantly) by each of the q processors. The eigenvalue refinement using bisection or Rayleigh quotient iteration ($O(n)$ per eigenvalue) is then parallelized over the q processors, resulting in a parallel complexity $(n|\Gamma_c|/q) = O(nk/p)$. Since many subsets of processors may be handling internal nodes at the same time, the overall parallel complexity is $O(\frac{nk}{p} + n)$. Note that due to communication overheads in a practical implementation, we impose a threshold on $|\Gamma_c|$ before the refinement is performed in parallel; if $|\Gamma_c|$ is below this threshold, the computation is carried out redundantly on all the q processors. On heterogeneous parallel machines, the redundant computation on every processor will need to be replaced by computation on a designated processor followed by a broadcast of the computed results to the other participating processors [7].

Figure 4 gives a description of Algorithm PMR^3 according to the strategy outlined

```
ALGORITHM PMR$^3$($T, \Gamma_0, tol, p$).                    {executed by processor $p_s$}
  Input: $T$ is the given symmetric tridiagonal;
         $\Gamma_0$ is the index set of desired eigenpairs;
         $tol$ is the input tolerance for relative gaps, usually set to $10^{-3}$;
         $p$ is the number of processors that execute the algorithm.
 Output: $(\hat{\lambda}_j, \hat{\boldsymbol{v}}_j), j \in \Gamma_0$ are the computed eigenpairs.
1. Split $T$ into irreducible subblocks $T_1, T_2, \ldots, T_\ell$.
     For each subblock $T_i$, $i = 1, \ldots, \ell$, do:
       A. Choose $\mu_i$ such that $L_0 D_0 L_0^t = T_i + \mu_i I$ is a factorization that determines
          the desired eigenvalues and eigenvectors, $\lambda_j$ and $\boldsymbol{v}_j$, $j \in \Gamma_0$, to high
          relative accuracy.
       B. Compute the desired eigenvalues $\Lambda_i$ of $L_0 D_0 L_0^t$ to high relative accuracy
          by the dqds algorithm [24] or by parallel bisection using a differential
          quotient-difference transform. Let $\Gamma_i \subseteq \Gamma_0$ be the index set correspond-
          ing to $\Lambda_i$ that contains the desired eigenvalues.
     end for
2. Determine the subset $\Gamma_0^s \subseteq \Gamma_0$ of eigenvectors to be computed locally. Form a
     work queue $\bar{Q}$, and initialize it with all the subblocks $(L_i, D_i, \Gamma_i)$ containing
     eigenvectors to be computed locally, i.e., the subblocks for which $\Gamma_i \cap \Gamma_0^s \neq \emptyset$,
     with $i = 1, \ldots, \ell$.
3. While queue $\bar{Q}$ is not empty:
     I. Remove an element $(L, D, \Gamma)$ from the queue $\bar{Q}$. Partition the computed
        eigenvalues $\hat{\lambda}_j, j \in \Gamma$, into clusters $\Gamma_1, \ldots, \Gamma_h$ according to their relative
        gaps and the input tolerance $tol$.
     II. For each cluster $\Gamma_c$, $c = 1, \ldots, h$, perform the following steps:
         If $\Gamma_c \subseteq \Gamma_0^{p_s}$ then all eigenvectors in $\Gamma_c$ have to be computed locally.
                      The eigenvectors are computed by invoking MR$^3$_Vec(($L, D$,
                      $\Gamma_c$), $tol$).
         elseif $\Gamma_c \cap \Gamma_0^s = \emptyset$ then the cluster $\Gamma_c$ does not contain any eigenvec-
                      tor that needs to be computed locally. Discard the
                      cluster $\Gamma_c$.
         elseif $\Gamma_c \cap \Gamma_0^s \neq \emptyset$ then the cluster $\Gamma_c$ contains some eigenvectors to be
                      computed locally, and needs to be further fragmented
                      by the following steps.
                      • Pick $\tau_c$ near the cluster and compute $LDL^t -$
                        $\tau_c I = L_c D_c L_c^t$ using the dstqds transform;
                        see [18, sect. 4.1] for details.
                      • "Refine" the eigenvalues $\hat{\lambda} - \tau_c$ in the cluster
                        so that they have high relative accuracy with
                        respect to the computed $L_c D_c L_c^t$. Set $\hat{\lambda} \leftarrow (\hat{\lambda} -$
                        $\tau_c)_{refined}$ for all eigenvalues in the cluster.
                      • Add $(L_c, D_c, \Gamma_c)$ to the queue $\bar{Q}$.
         end if
```

FIG. 4. *Algorithm* PMR^3 *for parallel computation of a subset* Γ_0 *of eigenvalues and eigenvectors.*

above. In order to show how subblocks of T are handled by the parallel algorithm, we do not assume that T is irreducible. Each processor will compute k/p eigenvectors, assuming k is divisible by p. Once the eigenvalues are grouped according to the subblocks and sorted (per subblock), work is assigned to the processors in a block cyclic manner; i.e., processor p_0 is assigned eigenvectors $1, 2, \ldots, k/p$, processor p_1 is assigned eigenvectors $k/p + 1, \ldots, 2k/p$, and so on. Thus the memory requirement to store the eigenvectors is exactly $(n \cdot k/p)$ floating point numbers per processor. The extra workspace required is linear in n, so problems of large size can be tackled. To give an idea of the limits of the sequential implementation, the size n of the largest problem that can be solved (with $k = n$) on a computer equipped with 1.5 GBytes of memory is about 14,000 when all the eigenvectors are required, while to solve a problem of size 30,000 a computer should be equipped with about 7 GBytes of main memory.

As seen in Figure 4, the eigenvectors are computed by invoking the sequential algorithms Getvec or MR^3_Vec (which in turn invokes Getvec). In terms of the representation tree, each processor maintains a local work queue $\bar{Q}$ of nodes (possibly leaves) which collectively index a superset of the eigenvectors to be computed locally. Initially all the processors have a single node in the queue corresponding to the desired index set Γ_0. The representation tree is traversed in a breadth-first fashion to fragment the clusters until all the eigenvectors of a node are local. Fragmenting a cluster is equivalent to descending one level in the representation tree. Nodes that contain eigenvectors all of which are associated with other processors are removed from the local queue of the processor. Once a node contains only eigenvectors to be computed locally, the sequential algorithms Getvec or MR^3_Vec are invoked depending on the size of the cluster. Recall that there is a tree for each subblock.

A word about the initial eigenvalue computation. The dqds algorithm for computing the eigenvalues is very fast but, like the QR algorithm, is inherently sequential. Moreover, the dqds algorithm cannot be adapted to compute k eigenvalues in $O(nk)$ time, instead always requiring $O(n^2)$ computations. On the other hand the bisection algorithm is easily parallelized [3]; however, bisection is rather slow. Thus in a parallel implementation, it is often preferable to redundantly compute the eigenvalues on each processor unless p is large or unless only a small subset of the n eigenvalues is desired. A quick calculation reveals the decision procedure to decide whether to use bisection or dqds. Bisection is linearly convergent, and finds one additional bit of an eigenvalue at every iteration. Thus, computing the desired eigenvalues by parallel bisection on p processors requires approximately $52n|\Gamma_0|/p$ operations in IEEE double precision arithmetic. On the other hand, dqds requires about 3 iterations, on average, to compute each eigenvalue to full relative accuracy. But dqds cannot be parallelized, so it requires approximately $3n^2$ operations irrespective of Γ_0. As a result, in our implementation, we use the dqds algorithm to redundantly compute all eigenvalues on the p processors when $3n^2 \leq 52n|\Gamma_0|/p$, i.e., $p \leq 17|\Gamma_0|/n$; otherwise we use parallel bisection.

We now illustrate the parallel execution of the algorithm on the matrix of Figure 3, assuming we want to compute all 11 eigenvectors on 3 processors. In Figure 5 we have annotated the representation tree of Figure 3 to show how the tree is processed by the 3 processors. Initially the eigenvalues $\Lambda_0 = \{\lambda_1, \ldots, \lambda_{11}\}$ are computed. Then based on the relative gaps between eigenvalues each processor determines whether a cluster is to be computed locally, has to be fragmented, or has to be discarded. The labels p_0, p_1, p_2 on the root node denote that each of the 3 processors is involved in the computation.

Processor p_0 classifies the eigenvalues λ_1 through λ_4, but discards all the clusters (possibly singletons) from λ_5 to the end of the spectrum as they do not contain

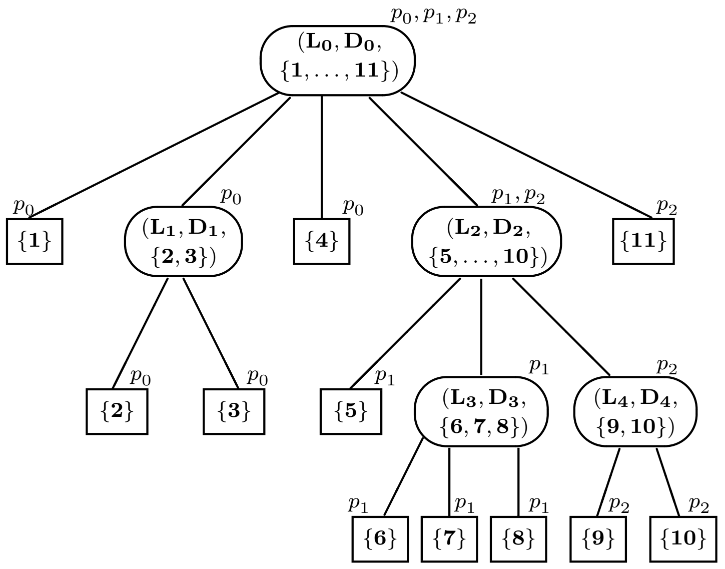
p_0, p_1, p_2
$(\mathbf{L_0}, \mathbf{D_0}, \{\mathbf{1}, \dots, \mathbf{11}\})$
p_0
$\{\mathbf{1}\}$
p_0
$(\mathbf{L_1}, \mathbf{D_1}, \{\mathbf{2}, \mathbf{3}\})$
p_0
$\{\mathbf{4}\}$
p_1, p_2
$(\mathbf{L_2}, \mathbf{D_2}, \{\mathbf{5}, \dots, \mathbf{10}\})$
p_2
$\{\mathbf{11}\}$
p_0
$\{\mathbf{2}\}$
p_0
$\{\mathbf{3}\}$
p_1
$\{\mathbf{5}\}$
p_1
$(\mathbf{L_3}, \mathbf{D_3}, \{\mathbf{6}, \mathbf{7}, \mathbf{8}\})$
p_2
$(\mathbf{L_4}, \mathbf{D_4}, \{\mathbf{9}, \mathbf{10}\})$
p_1
$\{\mathbf{6}\}$
p_1
$\{\mathbf{7}\}$
p_1
$\{\mathbf{8}\}$
p_2
$\{\mathbf{9}\}$
p_2
$\{\mathbf{10}\}$

FIG. 5. *Representation tree annotated to describe the execution of the parallel algorithm. The matrix size is* 11*; the algorithm is run on* 3 *processors.*

eigenvectors to be computed locally. The clusters $\{\lambda_1\}$, $\{\lambda_2, \lambda_3\}$, $\{\lambda_4\}$ contain eigenvalues local to p_0, so the corresponding nodes in the tree are labeled p_0. In classifying the eigenvalues, both processors p_1 and p_2 find that the cluster $\{\lambda_5, \dots, \lambda_{10}\}$ contains eigenvalues whose eigenvectors are to be computed locally: λ_5 through λ_8 for p_1 and λ_9, λ_{10} for p_2. The new representation is computed redundantly by both p_1 and p_2, and the refinement of eigenvalues λ_5 through λ_{10} is parallelized over p_1 and p_2. Thus the node is labeled p_1 as well as p_2 in Figure 5. The singleton λ_{11} is recognized as local by p_2 and therefore labeled p_2. The eigenvalue classification for node $(L_2, D_2, \{5, \dots, 10\})$ is independently performed by processors p_1 and p_2: p_1 recognizes the clusters $\{\lambda_5\}$, $\{\lambda_6, \lambda_7, \lambda_8\}$ to contain local eigenvalues and discards the cluster $\{\lambda_9, \lambda_{10}\}$; vice versa for processor p_2. Nodes $\{5\}$ and $(L_3, D_3, \{6, 7, 8\})$ are therefore labeled p_1 while node $(L_4, D_4, \{9, 10\})$ is labeled p_2.

It is important to realize that the parallel algorithm traverses the sequential representation tree in parallel. Assuming identical arithmetic on all processors, this implies that the computed eigenvectors match exactly the ones computed by the sequential algorithm and therefore satisfy the same accuracy properties.

3.2. Householder reduction and backtransformation. To solve the dense, symmetric eigenproblem the solution to the tridiagonal eigenproblem is preceded by reduction to tridiagonal form and followed by a backtransformation stage to obtain the eigenvectors of the dense matrix. We will see in the performance section that Algorithm PMR^3 discussed in the previous section reduces the cost of the tridiagonal eigenproblem sufficiently so that it is the reduction and backtransformation stages that dominate the computation time. In this section, we give a brief overview of the major issues behind the parallel implementation of these stages. A more detailed discussion can be found in [4].

Reduction to tridiagonal form is accomplished through the application of a sequence of orthogonal similarity transformations; usually Householder transformations are preferred to Givens rotations. At the ith step in the reduction, a Householder transformation is computed that annihilates the elements in the ith column that lie below the first subdiagonal. This transformation is then applied to the matrix from the left and the right, after which the computation moves on to the next column of the updated matrix. Unfortunately, this simple "unblocked" algorithm is rich in matrix-vector operations (matrix-vector multiplications and symmetric rank-one updates, to be exact) which do not achieve high performance on modern microprocessors. Thus, a blocked version of the algorithm is derived from the unblocked algorithm by delaying updates to the matrix, accumulating those updates into a so-called symmetric rank-k update [22]. This casts the computation in terms of matrix-matrix multiplication, which can achieve much better performance. However, it is important to note that even for the blocked algorithm, approximately half the computation is in symmetric matrix-vector multiplication. This means that the best one can hope for is that implementations based on the blocked algorithm improve performance by a factor of 2 over implementations based on the unblocked algorithm.

The backtransformation stage applies the Householder transforms encountered during the reduction to tridiagonal form to the eigenvectors computed for the tridiagonal matrix. It is well known how to accumulate such Householder transforms into block Householder transforms so that the computation is again cast in terms of matrix-matrix multiplication [6]. This time essentially all computation involves matrix-matrix multiplication, allowing very high performance to be achieved.

Parallel implementation of both these stages now hinges on the fact that the parallel implementation of the symmetric matrix-vector multiplication, the symmetric rank-k update, and matrix-matrix multiplication is scalable, and can achieve high performance. Since these issues are well understood, we omit presenting them here and refer the reader to [32, 31]. Some subtle differences in the parallel implementations of these stages as supported by ScaLAPACK and PLAPACK are given in the appendix of [4]. Essentially, the ScaLAPACK and PLAPACK implementations are tuned for smaller and larger matrices, respectively.

4. Experimental results.

This section presents timing results for the proposed algorithm. First we report results on the dense problem in section 4.2: it will be apparent that very large problems can now be tackled and that the tridiagonal eigenproblem is an order of magnitude faster than the reduction and backtransformation stages. In section 4.3 we focus on the tridiagonal eigensolvers and show that Algorithm PMR^3 achieves the best performance compared to previous algorithms.

4.1. Implementation details and test matrices.

All experiments were conducted on a cluster of Linux workstations. Each node in the cluster consisted of a dual Intel Pentium 4 processor (2.4 GHz) with 2 GBytes of main memory. The nodes were connected via a high-performance network (2 Gigabit/s) from Myricom. In our experiments, only one processor per node was enabled. The reason for using one processor was primarily related to the fact that during early experiments it was observed that reliable timings were difficult to obtain when both processors were enabled. Notice that the qualitative behavior of the different algorithms and implementations is not affected by this decision, even if the quantitative results are.

We will often refer to our proposed parallel dense eigensolver as Dense PMR^3, and use PMR^3 to denote the tridiagonal eigensolver outlined in Figure 4; however, sometimes we just use PMR^3 when it is clear whether we are referring to the dense or tridiagonal eigensolver. Dense PMR^3 has been implemented using the PLAPACK library for Householder reduction and backtransformation, while PMR^3 has been implemented in C and Fortran using the MPI library for communications and LAPACK for numerical routines.

We compare dense PMR^3 with the ScaLAPACK implementations of (a) bisection and inverse iteration (routine PDSYEVX) and (b) divide and conquer (routine PDSYEVD), and the PLAPACK implementation of (c) the QR algorithm (routine PLA_VDVt). All the routines have been compiled with the same optimization flags enabled and linked to the same high-performance BLAS library (the so-called GOTO BLAS, which in our experience achieve the highest performance on this machine [29]).

All dense eigensolvers have been tested on symmetric matrices of sizes ranging from 8000 to 128,000 with given eigenvalue distributions. We considered 4 types of diverse eigenvalue distributions:

1. UNIFORM (ε to 1):
$$\lambda_i = \varepsilon + (i-1) * \tau, \qquad i = 1, 2, \ldots, n,$$
where $\tau = (1-\epsilon)/(n-1)$.
2. GEOMETRIC (ε to 1):
$$\lambda_i = \varepsilon^{(n-i)/(n-1)}, \qquad i = 1, 2, \ldots, n.$$
3. RANDOM (ε to 1): the eigenvalues are drawn from a uniform distribution on the interval $[0, 1]$.
4. CLUSTERED at ε:
$$\lambda_1 \approx \lambda_2 \approx \cdots \approx \lambda_{n-1} \approx \varepsilon \ \text{ and } \ \lambda_n = 1.$$

In addition to the above "constructed" matrices, we also report timings for matrices arising in applications. We considered three matrices from computational quantum chemistry of sizes 966, 1687, and 2053, occurring, respectively, in modeling of the biphenyl molecule, study of bulk properties for the $\mathrm{SiOSi_6}$ molecule, and solution of a nonlinear Schrödinger problem using the self-consistent Hartree–Fock method for the zeolite ZSM-5. More details on these matrices can be found in [2, 13].

We also considered three matrices (sizes 7923, 12387, 13786) that arise in frequency response analyses of automobile bodies. These matrices come from a symmetric matrix pencil arising from a finite element model of order 1 million or so, going through a process of dividing the entire structure into several thousand "substructures" using nested dissection and finding the "lowest" eigenvectors for each substructure. Projecting the matrix pencil onto the substructure eigenvector subspace and then converting to standard form followed by Householder reduction yields the test tridiagonal matrices. Details on producing these matrices can be found in [35].

Notice that the matrices for which we report results are at least one order of magnitude larger than the results reported in [43, 32].

4.2. Results for the dense problem. We now present performance results for computing all eigenpairs of a dense symmetric matrix highlighting the difference between the $O(n^3)$ reduction and backtransformation stages and the $O(n^2)$ tridiagonal computation of PMR^3.

When possible we compare the proposed algorithm against the ScaLAPACK implementation of divide and conquer (PDSYEVD) [43] since the latter routine is the fastest among the tridiagonal eigensolvers currently available in ScaLAPACK. All matrices considered in the following results have random distribution of eigenvalues, i.e.,

TABLE 1
Timings in seconds for different stages of the dense eigensolvers, $n = 8000$, random eigenvalue distribution.

Stage	Method	# of processors				
		2	4	8	16	32
Reduction	PLAPACK	549	289	156	98	64
	ScaLAPACK	_	_	155	85	50
Tridiagonal	PMR^3	**13**	**9.4**	**7.4**	**6.3**	**3.9**
	PDSTEDC	_	_	32.3	19	11.3
Backtransformation	PLAPACK	178	95	51	29	18
	ScaLAPACK	_	_	113	65	34

TABLE 2
Timings in seconds for different stages of the dense eigensolvers, $n = 15000$, random eigenvalue distribution.

Stage	Method	# of processors			
		8	16	32	64
Reduction	PLAPACK	996	546	340	257
	ScaLAPACK	_	487	263	128
Tridiagonal	PMR^3	**26.2**	**22.2**	**13.6**	**6.9**
	PDSTEDC	_	92.4	52.0	32.5
Backtransformation	PLAPACK	292	160	93	65.7
	ScaLAPACK	_	226	114	64.2

matrices designated RANDOM in section 4.1. Note that neither the reduction nor the backtransformation stage is affected by the distribution of eigenvalues in the input matrix. Comparisons with other tridiagonal eigensolvers (QR algorithm, bisection, and inverse iteration) on matrices with varying eigenvalue distributions are given in section 4.3.

In Tables 1 and 2 we report timings for matrices of sizes 8000 and 15,000, respectively. The stages of dense PMR^3 are labeled PLAPACK (for reduction and backtransformation) and PMR^3 (for the tridiagonal solution), while the stages for the routine PDSYEVD are similarly labeled ScaLAPACK and PDSTEDC (the tridiagonal divide and conquer routine). As mentioned in section 2, a major drawback of the divide and conquer algorithm is its extra $O(n^2)$ memory requirement. As a result, there are several instances where PMR^3 can be run on a particular matrix, but PDSTEDC cannot be run; the symbol "_" in the tables indicates that the experiment could not be run because of memory constraints.

Figure 6 gives a pictorial view of the dense PMR^3 timings in Table 2. It is easy to see from the figure that the tridiagonal stage is an order of magnitude faster than the reduction and backtransformation stages. As justified by the calculations in section 3.1.3, Algorithm PMR^3 uses dqds for the initial eigenvalue computation when $p \leq 17|\Gamma_0|/n$; here $|\Gamma_0| = n$, so dqds is used when $p \leq 17$; otherwise parallel bisection with a differential quotient-difference transform is used.

Next we present timings for an incremental test of dense PMR^3, keeping the memory allocation per processor constant while increasing the matrix size and the number of processors accordingly. To compute all eigenvectors, the memory requirement is quadratic in n; hence for fixed memory per processor when the matrix size is doubled, four times as many processors are needed. So we give results for dense PMR^3

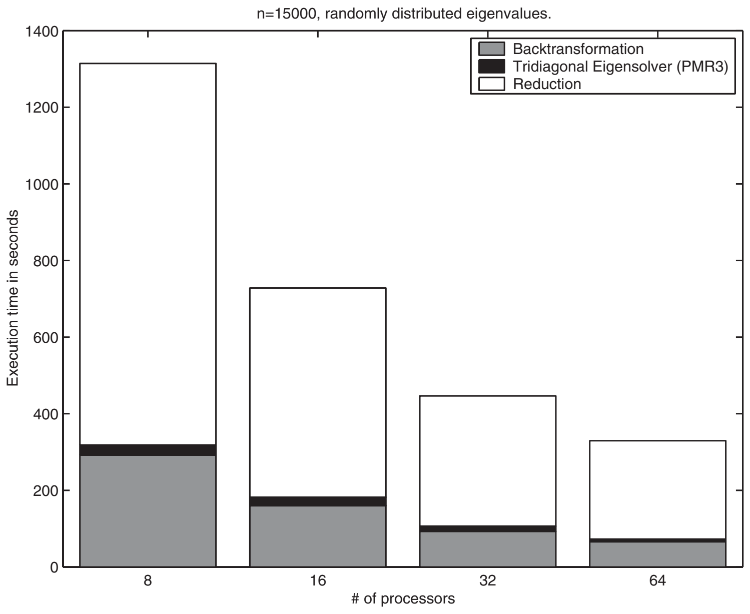

FIG. 6. *Timing breakdown for the dense* PMR^3 *algorithm,* $n = 15{,}000$.

TABLE 3
Timings for the dense eigensolvers. Columns 3 to 6 give timings for dense PMR^3*, while the last column reports extrapolated* (*) *numbers for a PLAPACK implementation of the QR algorithm. The numbers within parentheses in the fourth column* (PMR^3) *represent the time spent to compute eigenvalues. Entries in the last two columns are expressed in minutes.*

n	p	Reduction	PMR^3	Backtransformation	Total (dense PMR^3)	PLAPACK QR
8000	1	1081 s.	17.6 s. (5.4 s.)	348 s.	**24.1 min.**	100 min.*
16,000	4	2425 s.	38.8 s. (20.8 s.)	684 s.	**52.4 min.**	215 min.*
32,000	16	4876 s.	118 s. (97 s.)	1388 s.	**106.3 min.**	420 min.*
64,000	64	9638 s.	124 s. (104 s.)	2846 s.	**210.1 min.**	750 min.*
128,000	256	22,922 s.	128 s. (107 s.)	5827 s.	**481.3 min.**	

on matrices of sizes 8000, 16,000, 32,000, 64,000, and 128,000 using 1, 4, 16, 64, and 256 processors, respectively. Table 3 includes these timings in addition to extrapolated timings for the PLAPACK QR implementation.[1] We are unable to run divide and conquer (PDSYEVD) for this test, as it runs out of memory. The difference in complexity between the $O(n^2)$ tridiagonal eigensolver versus the $O(n^3)$ reduction and backtransformation is again obvious.

Indicating by $T_p(n)$ the total time from Table 3 to solve a problem of size n with p processors, we plot in Figure 7 the *incremental speedup*: the ratio $T_p(n\sqrt{p})/T_1(n)$ with $p = 1, 4, 16, 64, 256$. Since the dense eigensolver has $O(n^3)$ complexity, doubling the matrix size and deploying four times as many processors, assuming constant

[1]The performance of the QR algorithm applied to tridiagonal matrices is very predictable; the extrapolated timings have been obtained by running the tridiagonal problem with a larger number of processors (thereby avoiding memory problems) and adjusting the times for reduction and backtransformation, assuming perfect parallelization.

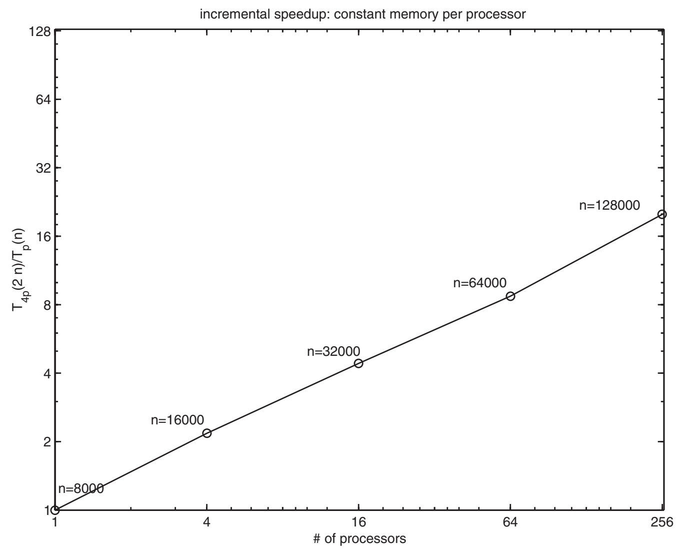

FIG. 7. *Incremental speedup.*

TABLE 4

Orthogonality and residual results for dense PMR^3*. Given the dense matrix A, the computed eigenvalues $\hat{\Lambda}$, and the computed eigenvectors $\hat{V}$, we display here* $\max_{i,j} |\hat{V}^T\hat{V} - I|_{ij}$ *(orthogonality) and* $\max_{i,j} |A\hat{V} - \hat{V}\hat{\Lambda}|_{ij}$ *(residual).*

Size		RANDOM	UNIFORM	GEOMETRIC	CLUSTERED
8000	Orthogonality	7.7e-11	2.0e-11	7.7e-12	2.5e-15
	Residual	3.7e-11	1.1e-11	1.5e-13	8.4e-16
15,000	Orthogonality	1.2e-10	1.0e-11	1.5e-11	1.7e-15
	Residual	1.9e-10	5.2e-12	6.8e-14	2.3e-16

performance per processor, the ratio $T_{4p}(2n)/T_p(n)$ equals

$$\frac{T_{4p}(2n)}{T_p(n)} = \frac{2^3\ \frac{T_p(n)}{4}}{T_p(n)} = \frac{8}{4} = 2.$$

The log-log graph in Figure 7 testifies that the theoretical prediction is almost perfectly matched in practice. This demonstrates that dense PMR^3 scales up well in performance and larger problems can be tackled effectively when more processors are available.

Finally, in Table 4 we report on the accuracy of Algorithm PMR^3. Note that the eigenvalues and eigenvectors returned by PMR^3 are exactly the same as those computed by the sequential algorithm; therefore they satisfy the properties for residual and orthogonality described in [17].

4.3. Results for the tridiagonal problem. We now focus on the tridiagonal stage, comparing the performance of three different algorithms on matrices with differing eigenvalue distributions. With earlier QR or inverse iteration–based algorithms, the tridiagonal eigenproblem is often the bottleneck in the solution of the dense eigenproblem. For example, running sequential code for a matrix of size 2000 with uniformly distributed eigenvalues, the reduction and backtransformation stages take a total of about 32 seconds, while the tridiagonal eigenproblem takes 86 seconds

TABLE 5
Timings in seconds for tridiagonal eigensolvers, $n = 8000$.

Distribution	Method	# of processors					
		1	2	4	8	16	32
RANDOM	PMR^3	**17.4**	**13**	**9.4**	**7.4**	**6.2**	**3.9**
	PDSTEDC	_	_	_	32.3	19	11.3
	PLA_VDVt	*	2152	1117	532	273	142
	PDSTE(BZ+IN)	*	*	*	5844	5817	5656
UNIFORM	PMR^3	**15.3**	**12.4**	**9.1**	**7.4**	**6.5**	**3.8**
	PDSTEDC	_	_	_	35	20.4	12.2
	PLA_VDVt	*	*	1115	536	275	143
	PDSTE(BZ+IN)	*	*	*	143	141	132
GEOMETRIC	PMR^3	**11**	**9.4**	**6.3**	**4.9**	**4.1**	**3.5**
	PDSTEDC	_	_	_	16.7	9.9	6.4
	PLA_VDVt	*	*	945	458	247	128
	PDSTE(BZ+IN)	*	*	*	5297	5348	5233
CLUSTERED	PMR^3	**.101**	**.06**	**.051**	**.043**	**.04**	**.037**
	PDSTEDC	_	_	_	2.7	1.4	1.0
	PLA_VDVt	*	*	888	450	229	122
	PDSTE(BZ+IN)	*	*	*	6293	6501	6373

for the QR algorithm, 106 seconds for bisection and inverse iteration, 4 seconds for the divide and conquer algorithm, but only .8 seconds for MR^3. The timings for a matrix of size 4000 are 290 seconds for reduction and backtransformation, 1099 seconds for QR, 821 for bisection and inverse iteration, 24 seconds for divide and conquer, and 3.1 seconds for MR^3. Note the $O(n^2)$ behavior of MR^3 as compared to the other tridiagonal methods. The above timings use the optimized GOTO BLAS; however, it may not always be possible to obtain optimized BLAS, especially for newer CPUs. The timings for $n = 4000$ using Fortran BLAS are 1125 seconds for QR, 830 for bisection and inverse iteration, 176 seconds for divide and conquer, and 3.4 seconds for MR^3. Thus, divide and conquer suffers the most when using Fortran BLAS.

In the following we illustrate the parallel performance of Algorithm PMR^3. Tables 5 and 7 include the timings for PMR^3 when the number of processors and eigenvalue distributions are varied. We compare our results with ScaLAPACK's divide and conquer routine (PDSYEVD) whenever possible; again the symbol "_" in the table indicates that the experiment could not be run because of memory constraints. We also present timings for a PLAPACK implementation of QR (PLA_VDVt); we found the ScaLAPACK QR implementation (PDSYEV) to be slower than the equivalent algorithm implemented with PLAPACK, probably due to synchronization issues; for this reason we omit PDSYEV timings. Finally for the experiment on matrices of size 8000 we also show timings for the ScaLAPACK implementation of bisection and inverse iteration (PDSTEBZ+PDSTEIN). An asterisk indicates that the experiment has not been run because of excessive time needed.

Table 5 shows that for matrices of order 8000 the PMR^3 algorithm is faster than all the other algorithms. In particular it is several orders of magnitude faster than bisection and inverse iteration; notice that the performance of the latter does not improve when increasing the number of processors. This is due to the presence of a large cluster of eigenvalues as judged by PDSTEIN. Since PDSTEIN does not split clusters across processors, all eigenvectors of the large cluster end up being computed

Table 6
Time spent in the computation of the eigenvectors by PMR^3, $n = 8000$.

Distribution	Time (in seconds) to compute eigenvectors					
	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
RANDOM	12	7.6	4.0	2.0	1.0	.5
UNIFORM	9.5	6.6	3.4	1.7	.9	.4
GEOMETRIC	7.8	6.0	3.0	1.6	.8	.4
CLUSTERED	.1	.059	.05	.042	.039	.037

Table 7
Timings in seconds for tridiagonal eigensolvers, $n = 15{,}000$.

Distribution	Method	# of processors			
		8	16	32	64
RANDOM	PMR^3	**26.2**	**22.2**	**13.6**	**6.9**
	PDSTEDC	-	92.4	52.0	32.5
	PLA_VDVt	$*$	2054	1000	565
UNIFORM	PMR^3	**26.4**	**22.7**	**13.6**	**6.8**
	PDSTEDC	-	101.5	56.8	35.3
	PLA_VDVt	$*$	2172	991	564
GEOMETRIC	PMR^3	**18.4**	**15.0**	**13.2**	**6.7**
	PDSTEDC	-	45.3	27.2	16.9
	PLA_VDVt	$*$	1890	839	467
CLUSTERED	PMR^3	**.16**	**.1**	**.1**	**.09**
	PDSTEDC	-	4.9	3.5	1.8
	PLA_VDVt	$*$	1448	827	440

on a single processor. Since PDSTE(BZ+IN) is not competitive, we do not present its results in Table 7. Routine PLA_VDVt is also several orders of magnitude slower than PMR^3, but it achieves perfect speedup and does not suffer from the memory problems of PDSTEDC.

Algorithm PMR^3 attains good performance, but for $p \leq 16$ it does not exhibit particularly good speedups due to the redundant serial eigenvalue computation. It is interesting to analyze the timings for just the eigenvector computations; we display these values in Table 6. Notice that the eigenvector computations attain very good speedups (the timings given for $p = 1$ are for the serial code, not the parallel code with $p = 1$). The exception is the case of clustered eigenvalues, for which the entire computation is extremely fast independent of the number of processors.

Results for matrices of size 15,000 are displayed in Table 7. The proposed algorithm is again the fastest in all the experiments. Finally, we consider six tridiagonal matrices coming from applications. Timings for PMR^3 are shown in Table 8; the parallel algorithm is again seen to be very fast; for example, it takes only $5 + 0.7$ seconds for a matrix of size 13,786 on 64 processors. Algorithm PMR^{3}achieves good speedups for the eigenvector computation in all cases, and the speedups for the eigenvalue computation are almost perfect when $p \geq 16$, i.e., when parallel bisection is used.

5. Conclusions.

In this paper, we have presented a new parallel eigensolver for computing all or a subset of the eigenvectors of a dense symmetric matrix. The tridiagonal kernel is not only faster than previous algorithms, but also scales up well in memory requirements ($O(n)$ work space), allowing for very large eigenproblems to be solved. As a result, the time now spent in the tridiagonal eigenproblem is negligible

TABLE 8
Timings in seconds for PMR^3 *on matrices from applications. Biphenyl, SiOSi$_6$, and ZSM-*5 *are matrices from quantum chemistry, while the matrices* auto.7923*,* auto.12387*, and* auto.13786 *arise in the finite element modeling of automobile bodies.*

Matrix	Size	Time to compute eigenvalues			Time to compute eigenvectors						
		$p = 1, 2, 4, 8, 16$	$p = 32$	$p = 64$	$p = 1$	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$	$p = 64$
Biphenyl	966	.072	.064	.035	.133	.091	.05	.027	.014	.007	.004
SiOSi$_6$	1687	.235	.17	.09	.436	.274	.155	.085	.043	.022	.011
ZSM-5	2053	.339	.241	.12	.752	.445	.228	.122	.066	.033	.017
auto.7923	7923	4.3	3.4	1.75	11.2	7.3	3.9	2.0	1.0	.52	.26
auto.12387	12,387	15.8	8.3	4.2	31.6	20.0	10.2	5.2	2.6	1.3	.65
auto.13786	13,786	13.5	9.8	5.0	31.8	22.3	11.3	5.7	2.8	1.4	.7

compared to the stages of Householder reduction and backtransformation. Thus, the onus is now squarely on speeding up the latter two stages if the dense symmetric eigensolver is to be sped up further.

Acknowledgments. We are indebted to the Center for Computational Research at the University at Buffalo, SUNY, for the use of their 300 compute node Dell Linux Cluster. We also acknowledge the Texas Advanced Computing Center (TACC) for use of their 32 compute node IBM Pentium III Linux Cluster. We thank George Fann for providing the matrices from computational quantum chemistry, and Jeff Bennighof and Mintae Kim for the matrices arising from frequency analyses of automobile bodies.

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*, 2nd ed., SIAM, Philadelphia, 1995.

[2] D. Bernholdt and R. Harrison, *Orbital invariant second order many-body perturbation on parallel computers: An approach for large molecules*, J. Chem. Physics, 102 (1995), pp. 9582–9589.

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

[4] P. Bientinesi, I. S. Dhillon, and R. van de Geijn, *A Parallel Eigensolver for Dense Symmetric Matrices Based on Multiple Relatively Robust Representations*, Technical report TR-03-26, Department of Computer Sciences, University of Texas, Austin, TX, 2003.

[5] C. Bischof, S. Huss-Lederman, X. Sun, A. Tsao, and T. Turnbull, *Parallel performance of a symmetric eigensolver based on the invariant subspace decomposition approach*, in Proceedings of the Scalable High Performance Computing Conference, Knoxville, TN, 1994.

[6] C. Bischof and C. Van Loan, *The WY representation for products of Householder matrices*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. s2–s13.

[7] L. Blackford, A. Cleary, J. Demmel, I. S. Dhillon, J. Dongarra, S. Hammarling, A. Petitet, H. Ren, K. Stanley, and R. Whaley, *Practical experience in the numerical dangers of heterogeneous computing*, ACM Trans. Math. Software, 23 (1997), pp. 133–147.

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

[9] R. P. Brent, *Algorithms for Minimization without Derivatives*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

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

[11] T. J. Dekker, *Finding a zero by means of successive linear interpolation*, in Constructive Aspects of the Fundamental Theorem of Algebra, B. Dejon and P. Henrici, eds., Wiley-Interscience, New York, 1969, pp. 37–48.

[12] J. Demmel and K. Veselić, *Jacobi's method is more accurate than QR*, SIAM J. Matrix Anal. Appl., 13 (1992), pp. 1204–1245.

[13] I. S. Dhillon, *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*, Ph.D. thesis, Computer Science Division, University of California, Berkeley, CA, 1997; available as UC Berkeley Technical report UCB//CSD-97-971.

[14] I. S. Dhillon, *Current inverse iteration software can fail*, BIT, 38 (1998), pp. 685–704.

[15] I. S. Dhillon, G. Fann, and B. N. Parlett, *Application of a new algorithm for the symmetric eigenproblem to computational quantum chemistry*, in Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997), CD-ROM, SIAM, Philadelphia, 1997.

[16] I. S. Dhillon and A. N. Malyshev, *Inner deflation for symmetric tridiagonal matrices*, Linear Algebra Appl., 358 (2002), pp. 139–144.

[17] I. S. Dhillon and B. N. Parlett, *Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices*, Linear Algebra Appl., 387 (2004), pp. 1–28.

[18] I. S. Dhillon and B. N. Parlett, *Orthogonal eigenvectors and relative gaps*, SIAM J. Matrix Anal. Appl., 25 (2004), pp. 858–899.

[19] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling, *A set of level* 3 *basic linear algebra subprograms*, ACM Trans. Math. Software, 16 (1990), pp. 1–17.

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

[21] J. Dongarra, R. van de Geijn, and D. Walker, *A look at scalable dense linear algebra libraries*, in Proceedings of the Scalable High-Performance Computing Conference, IEEE Computer Society Press, Los Alamitos, CA, 1992, pp. 372–379.

[22] J. Dongarra, S. J. Hammarling, and D. C. Sorensen, *Block reduction of matrices to condensed forms for eigenvalue computations*, J. Comput. Appl. Math., 27 (1989), pp. 215–227.

[23] D. Elwood, G. Fann, and R. Littlefield, *PeIGS User's Manual*, Pacific Northwest National Laboratory, Richland, WA, 1993.

[24] K. Fernando and B. Parlett, *Accurate singular values and differential qd algorithms*, Numer. Math., 67 (1994), pp. 191–229.

[25] K. V. Fernando, *On computing an eigenvector of a tridiagonal matrix. Part* I: *Basic results*, SIAM J. Matrix Anal. Appl., 18 (1997), pp. 1013–1034.

[26] G. J. F. Francis, *The QR transformation: A unitary analogue to the LR transformation.* I, Comput. J., 4 (1961/1962), pp. 265–271.

[27] G. J. F. Francis *The QR transformation.* II, Comput. J., 4 (1961/1962), pp. 332–345.

[28] W. J. Givens, *Numerical Computation of the Characteristic Values of a Real Symmetric Matrix*, Technical report ORNL-1574, Oak Ridge National Laboratory, Oak Ridge, TN, 1954.

[29] K. Goto and R. A. van de Geijn, *On Reducing TLB Misses in Matrix Multiplication*, Technical report CS-TR-02-55, Department of Computer Sciences, University of Texas, Austin, TX, 2002; see also http://www.cs.utexas.edu/users/kgoto.

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

[31] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn, *A flexible class of parallel matrix multiplication algorithms*, in Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing, 1998, pp. 110–116.

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

[33] C. G. F. Jacobi, *Concerning an easy process for solving equations occurring in the theory of secular disturbances*, J. Reine Angew. Math., 30 (1846), pp. 51–94.

[34] E. R. Jessup and I. C. F. Ipsen, *Improving the accuracy of inverse iteration*, SIAM J. Sci. Statist. Comput., 13 (1992), pp. 550–572.

[35] M. F. Kaplan, *Implementation of Automated Multilevel Substructuring for Frequency Response Analysis of Structures*, Ph.D. thesis, University of Texas, Austin, TX, 2001.

[36] V. N. Kublanovskaya, *On some algorithms for the solution of the complete eigenvalue problem*, Zh. Vych. Mat., 1 (1961), pp. 555–570.

[37] T. Y. Li and Z. G. Zeng, *The Laguerre iteration in solving the symmetric tridiagonal eigenproblem, revisited*, SIAM J. Sci. Statist. Comput., 15 (1994), pp. 1145–1173.

[38] B. N. Parlett, *The Symmetric Eigenvalue Problem*, 2nd ed., SIAM, Philadelphia, 1997.

[39] B. N. Parlett, *Laguerre's method applied to the matrix eigenvalue problem*, Math. Comp., 18 (1964), pp. 464–485.

[40] B. N. Parlett and I. S. Dhillon, *Fernando's solution to Wilkinson's problem: An application of double factorization*, Linear Algebra Appl., 267 (1997), pp. 247–279.

[41] B. N. Parlett and I. S. Dhillon, *Relatively robust representations of symmetric tridiagonals*, Linear Algebra Appl., 309 (2000), pp. 121–151.

[42] G. Peters and J.H. Wilkinson, *The calculation of specified eigenvectors by inverse iteration, contribution* II/18, in Handbook of Automatic Computation, Vol. 2, Springer-Verlag, New York, Heidelberg, Berlin, 1971, pp. 418–439.

[43] F. Tisseur and J. Dongarra, *A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures*, SIAM J. Sci. Comput., 20 (1999), pp. 2223–2236.

[44] R. van de Geijn, *Using PLAPACK: Parallel Linear Algebra Package*, MIT Press, Cambridge, MA, 1997.

[45] J. H. Wilkinson, *The Algebraic Eigenvalue Problem*, Oxford University Press, Oxford, 1965.