A Parallel Eigensolver for Dense Symmetric Matrices based on Multiple Relatively Robust Representations

UT CS Technical Report # TR-03-26

Paolo Bientinesi, Inderjit S. Dhillon and Robert A. van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
{pauldj,inderjit,rvdg}@cs.utexas.edu

July 29, 2003

Abstract

We present a new parallel solution for the dense symmetric eigenvalue/eigenvector problem that is based upon the tridiagonal eigensolver, Algorithm MR^3, recently developed by Dhillon & 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 only requires $O(n)$ 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 & 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 problem 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 8 hours and 24 minutes 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.

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) 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 [41, 57], inverse iteration [67] and the Divide & Conquer method [18].

Indeed, 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 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 tridiagonalization 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 & Parlett proposed Algorithm MRRR or MR^3 (Algorithm of **M**ultiple **R**elatively **R**obust **R**epresentations) [24, 29, 28], 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 secs for reduction, 22.2 secs for the tridiagonal solution and 160 secs for backtransformation. In comparison, the corresponding timings for existing implementations for the tridiagonal eigensolution are: 2054 secs for the QR algorithm and 92.4 secs for the Divide & Conquer method. For a $32,000 \times 32,000$ matrix the timings for PMR^3 on 16 processors are: 4876 secs for the reduction, 118 secs for the tridiagonal solution and 1388 secs 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 existing algorithms have extra memory requirements: the ScaLAPACK Divide & 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 only requires 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 8 hours and 24 minutes of computation time. Detailed timing results are given in Section 4.

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 that 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. Conclusions are presented in the final section.

A word on the notation used throughout the paper. T indicates a tridiagonal matrix, n represents the size of a matrix, eigenvalues are denoted by λ and eigenvectors 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 i-th 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 are 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 [41] and Kublanovskaja [57], 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 [63]. A fast square-root free version of QR developed by Pal, Walker and Kahan (PWK) is useful if only eigenvalues are desired [63]. Another attractive alternative, in the latter case, is to use the differential qd algorithm (dqds) that is based on the related LR iteration [39]. 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 [27].

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 [50]. 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 [42]. Once accurate eigenvalues are known, the method of inverse iteration may be used to compute the corresponding eigenvectors [67]. However, inverse iteration can only guarantee 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 [25]. 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 [54]. Unfortunately the latter situation is the norm rather than the exception for large matrices since even uniform eigenvalue spacings when n exceeds 1,000 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 [11, p. 48].

The bisection algorithm to find eigenvalues has linear convergence and so, can be quite slow. To

speed up bisection, there have been many attempts to employ faster zero-finders such as the Rayleigh Quotient Iteration [63], Laguerre's method [55, 61, 64] and the Zeroin scheme [19, 13]. These zero-finders can speed up the computation of isolated eigenvalues by a considerable amount but they seem to stumble when eigenvalues cluster. Homotopy methods for the symmetric eigenproblem were suggested by Chu in [16, 17]. These methods start from an eigenvalue of a simpler matrix D and follow a smooth curve to find an eigenvalue of $A(t) \equiv D + t(A - D)$, varying t from 0 to 1. D was chosen to be the diagonal of the tridiagonal in [60], but greater success was obtained by taking D to be a direct sum of submatrices of T [62]. An alternate divide and conquer method that finds the eigenvalues by using Laguerre's iteration instead of homotopy methods is given in [61]. Note that in all these cases, the corresponding eigenvectors still need to be obtained by inverse iteration.

The Divide & Conquer method proposed by Cuppen in 1981 is a method specially suited for parallel computation [18, 33]; remarkably this algorithm also yields a faster sequential implementation than QR. The basic strategy of the Divide & 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 [14], or may be obtained by crossing out a row and column of the tridiagonal [43, 45]. 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 [46].

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 is 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 & Conquer method is suited for parallelization since smaller sub-problems can be solved independently on various processors. However, communication costs for combining sub-problems are substantial, especially when combining the larger sub-problems to get the solution to the full problem [72]. The biggest drawback of the Divide & Conquer algorithm is the extra $O(n^2)$ main memory required for its computation — 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 in 1846 [53], and was rediscovered by von Neumann and colleagues in 1946 [4]. Jacobi's method does not reduce the dense symmetric matrix to tridiagonal form, as most other methods do, but works on the dense matrix itself. 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 [23].

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 [12, 74, 7] and theoretical results show it to be more accurate than the QR algorithm [22].

The Symmetric Invariant Subspace Decomposition Algorithm (SYISDA) formulates the problem in a dramatically different way [9]. 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 this 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 [30, 77, 49, 44, 1, 75, 47]. Unfortunately, this approach increases 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 [21]. Routines for this problem have been developed as part of a number of numerical libraries. Among these the best known are the Scalable Linear Algebra Package (ScaLAPACK) [34, 11], Parallel Eigensolver (PeIGS) [37], the Parallel Research on Invariant Subspace Methods (PRISM) project [9], and the Parallel Linear Algebra Package (PLAPACK) [73, 2]. All of these packages attempt to achieve portability by embracing the Message-Passing Interface (MPI) [32] and the Basic Linear Algebra Subprograms (BLAS) [58, 31, 30].

The ScaLAPACK project is an effort to parallelize the Linear Algebra Package (LAPACK) [3] 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 currently includes a parallel tridiagonal eigensolver that is based on an early version of Algorithm MR^3; this preliminary version does limited Gram-Schmidt orthogonalization and was called the Berkeley algorithm in [26]. 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, 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 & Parlett [24, 29, 28] for the task of computing k eigenvectors of a symmetric tridiagonal 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 [24, 29, 28]. There are three key ingredients that form the backbone of Algorithm MR^3:

1. **Relatively Robust Representations(RRRs).** A relatively robust representation 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 [29, Sec. 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 [29]. 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

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

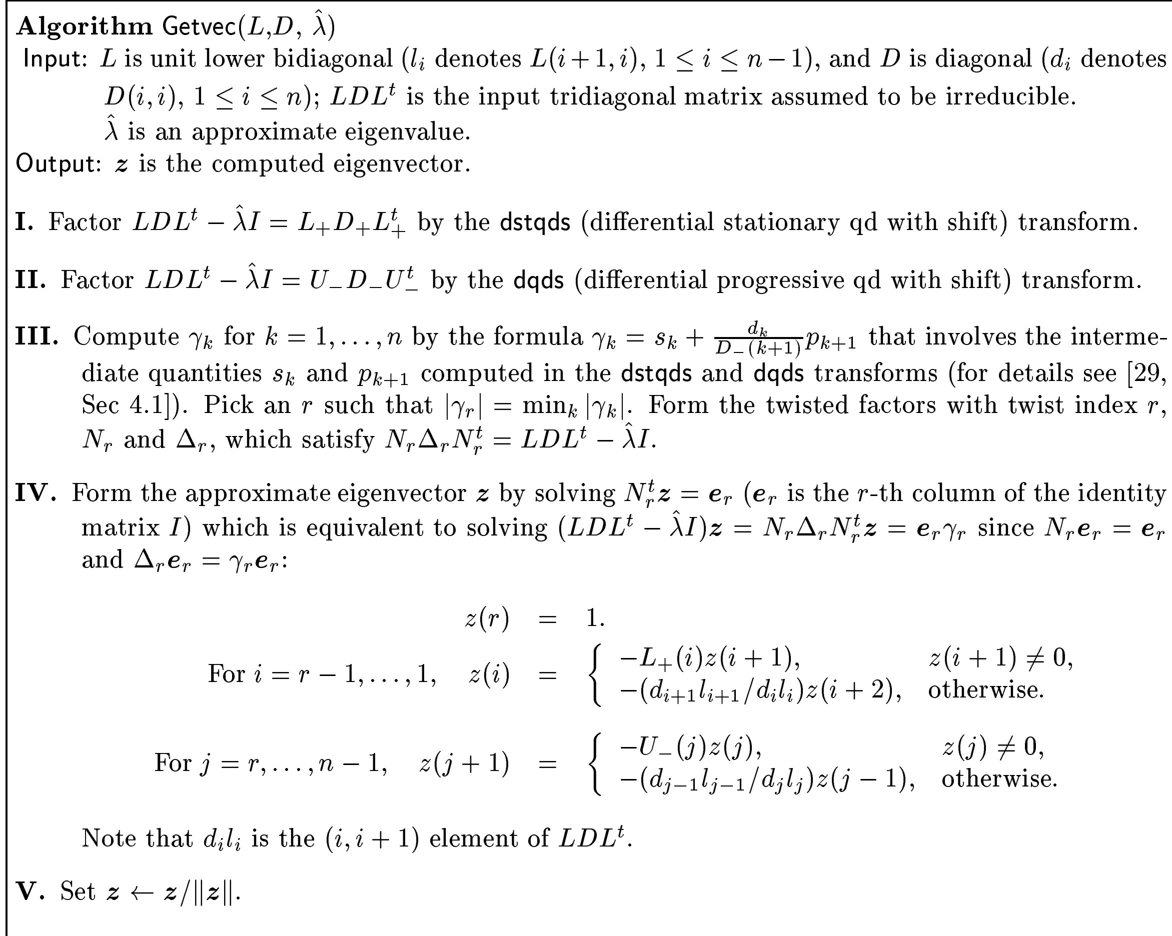
$$|\sin \angle(\boldsymbol{v}, \boldsymbol{v}+\delta\boldsymbol{v})| \leq \frac{K_2 n\xi}{\text{relgap}(\lambda)}, \tag{2}$$

for modest constants K_1 and K_2, say, smaller than 100. We call such an LDL^t factorization a relatively robust representation (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 [29].

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 [78, 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 [29, 65, 40] 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 [29]. However, when eigenvalues have small relative gaps, the above approach is not adequate. For the case of small relative gaps, Algorithm MR^3 uses multiple RRRs, i.e., multiple factorizations $L_c D_c L_c^t = LDL^t - \tau_c I$, 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 $L_c D_c L_c^t$, the eigenvalues in the cluster are "refined" so that they have high relative accuracy with respect to $L_c D_c L_c^t$. Finally the eigenvectors of eigenvalues that become relatively well separated after shifting are computed by Algorithm Getvec using $L_c D_c L_c^t$; 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 [66] and [28]. 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



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: $\boldsymbol{z}$ is the computed eigenvector.

I. Factor $LDL^t - \hat{\lambda} I = L_+ D_+ L_+^t$ by the dstqds (differential stationary qd with shift) transform.

II. Factor $LDL^t - \hat{\lambda} I = U_- D_- U_-^t$ by the dqds (differential progressive qd with shift) transform.

III. Compute γ_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 [29, Sec 4.1]). Pick an r such that $|\gamma_r| = \min_k |\gamma_k|$. Form the twisted factors with twist index r, N_r and Δ_r, which satisfy $N_r \Delta_r N_r^t = LDL^t - \hat{\lambda} I$.

IV. Form the approximate eigenvector $\boldsymbol{z}$ by solving $N_r^t \boldsymbol{z} = \boldsymbol{e}_r$ ($\boldsymbol{e}_r$ is the r-th column of the identity matrix I) which is equivalent to solving $(LDL^t - \hat{\lambda} I)\boldsymbol{z} = N_r \Delta_r N_r^t \boldsymbol{z} = \boldsymbol{e}_r \gamma_r$ since $N_r \boldsymbol{e}_r = \boldsymbol{e}_r$ and $\Delta_r \boldsymbol{e}_r = \gamma_r \boldsymbol{e}_r$:

$$
\begin{aligned}
z(r) &= 1. \\
\text{For } i = r-1, \ldots, 1, \quad z(i) &= \begin{cases} -L_+(i) z(i+1), & z(i+1) \neq 0, \\ -(d_{i+1} l_{i+1} / d_i l_i) z(i+2), & \text{otherwise.} \end{cases} \\
\text{For } j = r, \ldots, n-1, \quad z(j+1) &= \begin{cases} -U_-(j) z(j), & z(j) \neq 0, \\ -(d_{j-1} l_{j-1} / d_j l_j) z(j-1), & \text{otherwise.} \end{cases}
\end{aligned}
$$

Note that $d_i l_i$ is the $(i, i+1)$ element of LDL^t.

V. Set $\boldsymbol{z} \leftarrow \boldsymbol{z} / \|\boldsymbol{z}\|$.

Figure 1: Algorithm Getvec for computing the eigenvector of an isolated eigenvalue.

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 $(LDL^t - \hat{\lambda} I)\boldsymbol{z} = \gamma_r \boldsymbol{e}_r$; thus r is the index of the equation that is ignored and provides a solution to Wilkinson's problem (see above). 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 qd transforms and Algorithm Getvec may be found in [29].

Algorithm Getvec computes a single eigenvector of an RRR; it was shown in [29] 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 even 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

Algorithm MR^3(T,Γ_0,tol)
Input: T is the given symmetric tridiagonal,
Γ_0 is the index set of desired eigenpairs,
tol is the input tolerance for relative gaps, usually tol is 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 sub-blocks $T_1, T_2, \ldots, T_\ell$.
For each sub-block T_i $i = 1, \ldots, \ell$, do:
- **A.** Choose μ_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, λ_j and $\boldsymbol{v}_j$, $j \in \Gamma_0$, to high relative accuracy. In general, the shift μ_i can be in the interior of T's spectrum, but a safe choice is to make $T + \mu_i I$ 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 [39] or by bisection using a differential qd transform.
- **C.** Form a work queue Q, and initialize $Q = \{(L_0, D_0, \Gamma_0)\}$. Call MR^3_Vec(Q,tol).

end for

Subroutine MR^3_Vec(Q,tol)

While Queue Q is not empty:

I. Remove an element (L, D, Γ) from the queue 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. The eigenvalues are thus designated as isolated (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 non-trivial cluster Γ_c ($|\Gamma_c| > 1$) satisfy $|\hat{\lambda}_j - \hat{\lambda}_{j-1}|/|\hat{\lambda}_j| < tol$.

II. **For** each cluster Γ_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 Algorithm Getvec(L,D,$\hat{\lambda}_j$) to obtain the computed eigenvector $\hat{\boldsymbol{v}}_j$.

else

- **a.** Pick τ_c near the cluster and compute $LDL^t - \tau_c I = L_c D_c L_c^t$ using the dstqds (differential form of stationary qd) transform, see [29, Sec 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, Γ_c) to the queue Q.

end if

end for

end while

Figure 2: Algorithm MR^3 for computing orthogonal eigenvectors without using Gram-Schmidt orthogonalization.

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 [28]. 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 (L_p, D_p, Γ_p) be its parent node. If Π_c is a non-leaf 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. This node captures the fact 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 $\{\lambda_c\}$, where $c \in \Gamma_p$. The singleton node $\{\lambda_c\}$ 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.

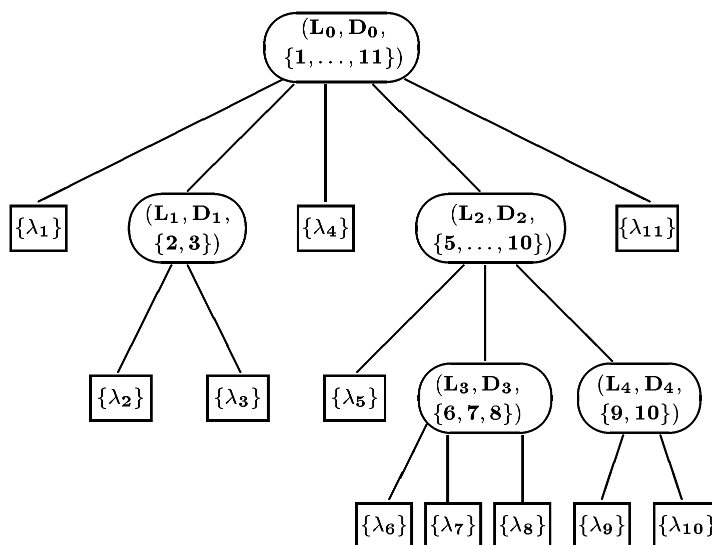


Figure 3: An example representation tree for a matrix of size 11.

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 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 $|(\hat{\lambda}_3 - \hat{\lambda}_2)/\hat{\lambda}_3| \geq tol$, 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 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 $\{\lambda_2\}$ and $\{\lambda_3\}$ in the first node are now relatively well separated, while the second node is further fragmented into a relatively well separated eigenvalue $\{\lambda_5\}$ 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.

In IEEE double precision arithmetic, using the tolerance $\min(10^{-3}, 1/n)$ for relative gaps, the depth of a representation tree can be as large as 6, see [28] for an example. In most examples we have encountered, the depth is much smaller — to give a sense of reality 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 ($n = 13786$) has maximum depth 2; at depth 1 there are 12937 singleton nodes, 403 clusters of size 2, 10 clusters of size 3 and one cluster of size 13. The tree for the matrix auto.12387 ($n = 12387$) 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 sub-blocks thus the sequence of computations can be captured by a forest of trees (one for each sub-block) rather than by a single tree.

3.1.3 The Parallel Algorithm

We now describe Algorithm PMR^3 (**P**arallel algorithm using **M**ultiple **R**elatively **R**obust **R**epresentations). 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 [32]. We assume that the tridiagonal is duplicated 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 $\{\lambda_i\}$ 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 qd 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.

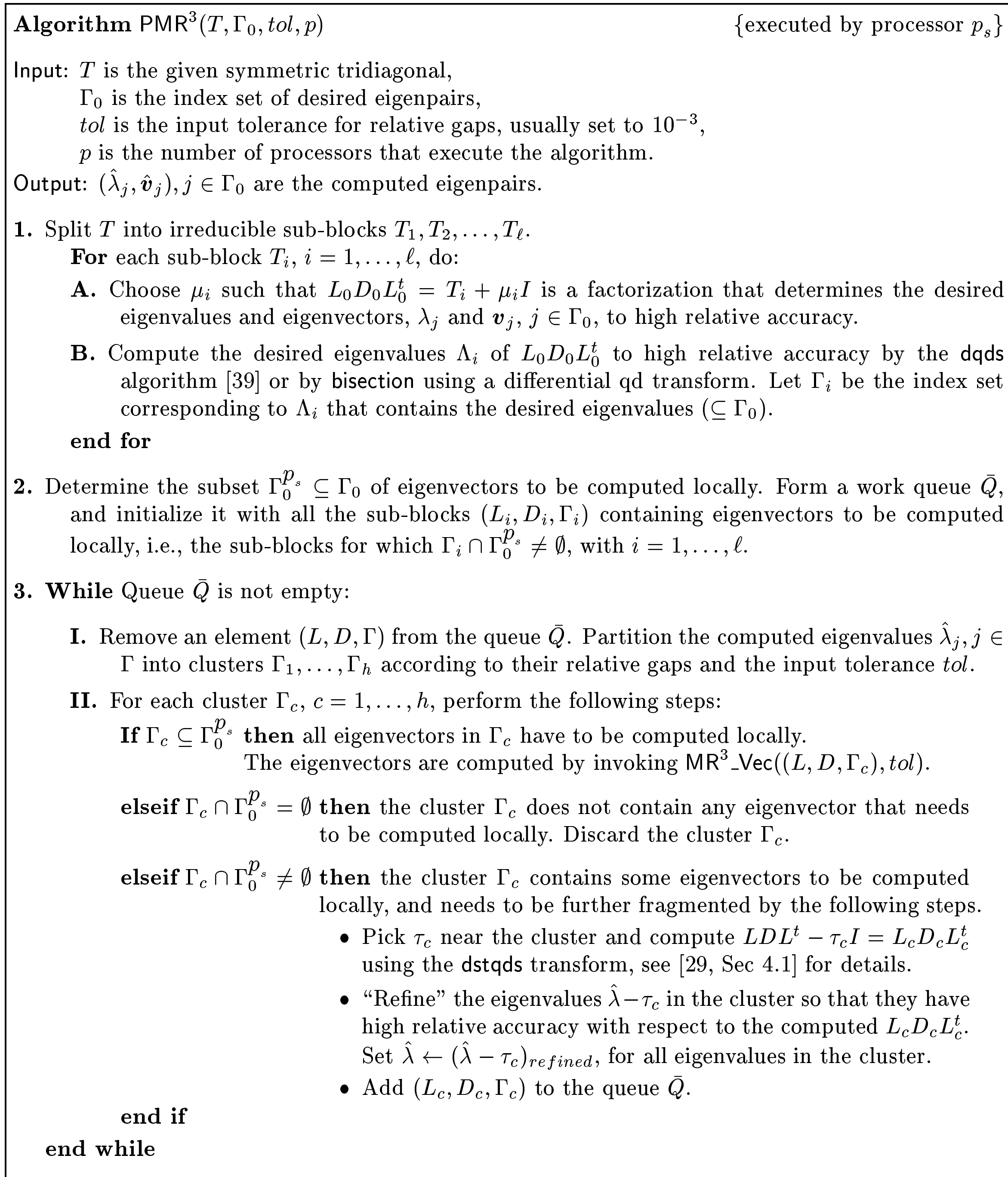
However, before the leaf nodes can be processed the computation at the internal nodes needs to be performed. Each internal node (L_c, D_c, Γ_c) is associated with a subset of q processors that are responsible for computing the eigenvectors in Γ_c. Since k eigenvectors are to be computed by a total of p processors, q approximately equals $p(|\Gamma_c|/k)$. 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 at a cost of $(n|\Gamma_c|/q) = O(nk/p)$ operations. Since many subsets of processors may be handling internal nodes at the same time, and the depth of the tree is at most 6, 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.

Figure 4 gives a description of Algorithm PMR^3 according to the strategy outlined above. In order to show how sub-blocks 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 sub-blocks and sorted (per sub-block), 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 only 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. To fragment 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 just contains 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 sub-block.

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 [6], 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 only a small subset of the n eigenvalues is desired.

We now illustrate the parallel execution of the algorithm on the matrix of Figure 3 assuming we want to compute all the 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.



Algorithm PMR^3(T, Γ_0, tol, p) $\{$executed by processor $p_s\}$

Input: T is the given symmetric tridiagonal,
Γ_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 sub-blocks $T_1, T_2, \ldots, T_\ell$.
For each sub-block T_i, $i = 1, \ldots, \ell$, do:
A. Choose μ_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, λ_j and $\boldsymbol{v}_j$, $j \in \Gamma_0$, to high relative accuracy.
B. Compute the desired eigenvalues Λ_i of $L_0 D_0 L_0^t$ to high relative accuracy by the dqds algorithm [39] or by bisection using a differential qd transform. Let Γ_i be the index set corresponding to Λ_i that contains the desired eigenvalues ($\subseteq \Gamma_0$).
end for

2. Determine the subset $\Gamma_0^{p_s} \subseteq \Gamma_0$ of eigenvectors to be computed locally. Form a work queue $\bar{Q}$, and initialize it with all the sub-blocks (L_i, D_i, Γ_i) containing eigenvectors to be computed locally, i.e., the sub-blocks for which $\Gamma_i \cap \Gamma_0^{p_s} \neq \emptyset$, with $i = 1, \ldots, \ell$.

3. While Queue $\bar{Q}$ is not empty:

I. Remove an element (L, D, Γ) 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 Γ_c, $c = 1, \ldots, h$, perform the following steps:
If $\Gamma_c \subseteq \Gamma_0^{p_s}$ **then** all eigenvectors in Γ_c have to be computed locally.
The eigenvectors are computed by invoking MR^3_Vec$((L, D, \Gamma_c), tol)$.

elseif $\Gamma_c \cap \Gamma_0^{p_s} = \emptyset$ **then** the cluster Γ_c does not contain any eigenvector that needs to be computed locally. Discard the cluster Γ_c.

elseif $\Gamma_c \cap \Gamma_0^{p_s} \neq \emptyset$ **then** the cluster Γ_c contains some eigenvectors to be computed locally, and needs to be further fragmented by the following steps.
- Pick τ_c near the cluster and compute $LDL^t - \tau_c I = L_c D_c L_c^t$ using the dstqds transform, see [29, Sec 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, Γ_c) to the queue $\bar{Q}$.

end if
end while

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

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 eigenvalues to be computed locally. The clusters $\{\lambda_1\}$, $\{\lambda_2, \lambda_3\}$, $\{\lambda_4\}$ contain eigenvalues local to p_0, so the nodes in the tree are labelled with p_0. In classifying the eigenvalues both processors p_1 and p_2 find that the cluster

$\{\lambda_5, \ldots, \lambda_{10}\}$ contains eigenvalues to be computed locally: λ_5 through λ_8 for p_1 and λ_9, λ_{10} for p_2. Thus, 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 labelled with both p_1 and p_2 in Figure 5. The singleton $\{\lambda_{11}\}$ is recognized as local by p_2 and therefore labelled with p_2. The eigenvalue classification for node $(L_2, D_2, \{5, \ldots, 10\})$ 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 $\{\lambda_5\}$ and $(L_3, D_3, \{6, 7, 8\})$ are therefore labelled p_1 while node $(L_4, D_4, \{9, 10\})$ is labelled p_2.

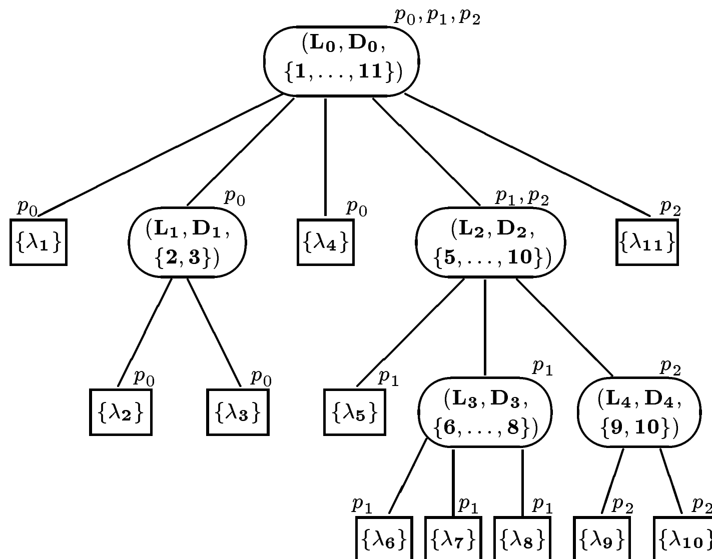


Figure 5: Representation tree annotated to describe the execution of the parallel algorithm. The matrix size is 11; the algorithm is executed by 3 processors.

It is important to realize that the parallel algorithm traverses the sequential representation tree in parallel. 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 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 algorithms. A more detailed discussion can be found in the appendices.

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 i-step in the reduction, a Householder transformation is computed that annihilates the elements in the i-th 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-1 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 [35]. 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 two 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 [3, 10]. 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 [50, 51, 59, 15, 73, 1, 75, 15, 47]. Some subtle differences in the parallel implementations of these stages as supported by ScaLAPACK and PLAPACK are given in the appendices. 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 showing 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 (R) 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. As explained later, we use the dqds algorithm for the initial eigenvalue computation (step B in Figure 4).

We compare Dense PMR^3 with the ScaLAPACK implementations of (a) Bisection and Inverse Iteration (routine `PDSYEVX`) and (b) Divide & 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 [44]).

All dense eigensolvers have been tested on symmetric matrices of sizes ranging from 8,000 to 128,000 with given eigenvalue distributions. We considered 4 types of eigenvalue distributions:

1. UNIFORM (ε to 1):
$$\lambda_i = \varepsilon + (i-1) * \tau, \quad i = 1, 2, \ldots, n$$
where $\tau = (1-\epsilon)/(n-1)$.

2. GEOMETRIC (ε to 1):
$$\lambda_i = \varepsilon^{(n-i)/(n-1)}, \quad i = 1, 2, \ldots, n.$$

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 $SiOSi_6$ molecule and solution of a non-linear Schrödinger problem using the self consistent Hartree-Fock method. More details on these matrices can be found in [5, 38].

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 [56].

Notice that the matrices for which we report results are at least one order of magnitude larger than the results reported in [72, 50].

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 & Conquer (`PDSYEVD`) [72] 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. 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 15000 respectively. The stages of Dense PMR^3 are labelled by PLAPACK or PMR^3, while the stages for the routine `PDSYEVD` are labelled by ScaLAPACK and `PDSTEDC` (the tridiagonal divide & conquer routine). As mentioned in Section 2, a major drawback of the Divide & Conquer algorithm is its extra $O(n^2)$ memory requirement. As a result, there are several instances where Dense 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. For PMR^3, we use the fast dqds algorithm for computing the eigenvalues with