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 28, 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 unprecedented size to be solved in a reasonable amount of time — 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 multivariate statistics. 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 now 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 PMR^3 takes 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. 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 solve problems of unprecedented size in a reasonable time. 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 given in the final section.

A word on the notation that is frequently 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, 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, as shown 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

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

where $\mathrm{gap}(\hat{\lambda}) = \min\{|\nu - \hat{\lambda}| : \nu \neq \lambda, \ \nu \in \mathrm{spectrum}(LDL^t)\}$. 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

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

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

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 the above method of computing eigenvectors, it can be proved 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 $L_c D_c L_c^t = LDL^t - \tau_c I$. 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] respectively. It is important to note that orthogonality of the computed eigenvectors is achieved without Gram-Schmidt being used in any of the procedures.

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.

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 $(LDL^t - \hat{\lambda}I)\boldsymbol{z} = \gamma_r \boldsymbol{e}_r$; 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 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 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 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 maximum possible depth of a representation tree is 6, see [28] for an example. 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

Algorithm $\mathsf{MR}^3(T,\Gamma_0,tol)$

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 $\min(10^{-3}, 1/n)$.

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.

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.

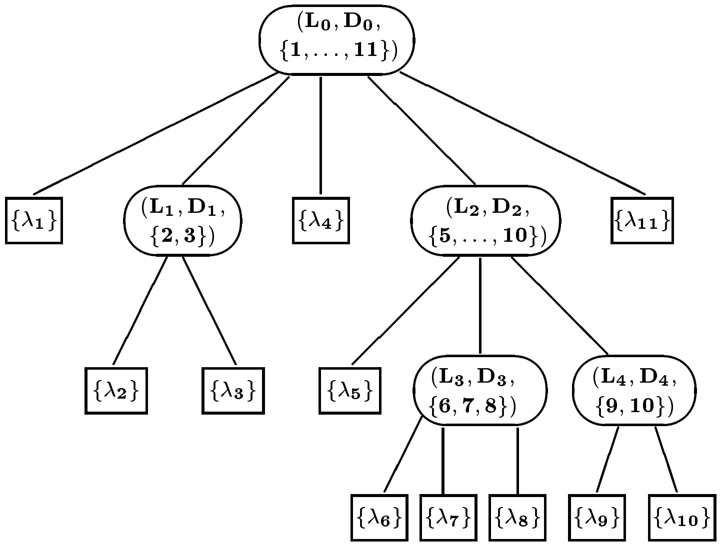

Figure 3: An example representation tree for a matrix of size 11.

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 $\mathsf{PMR}^3(T, \Gamma_0, tol, p)$ $\{$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 $\min(10^{-3}, 1/n)$,
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_$\mathsf{Vec}((L, D, \Gamma_c), tol)$.

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.

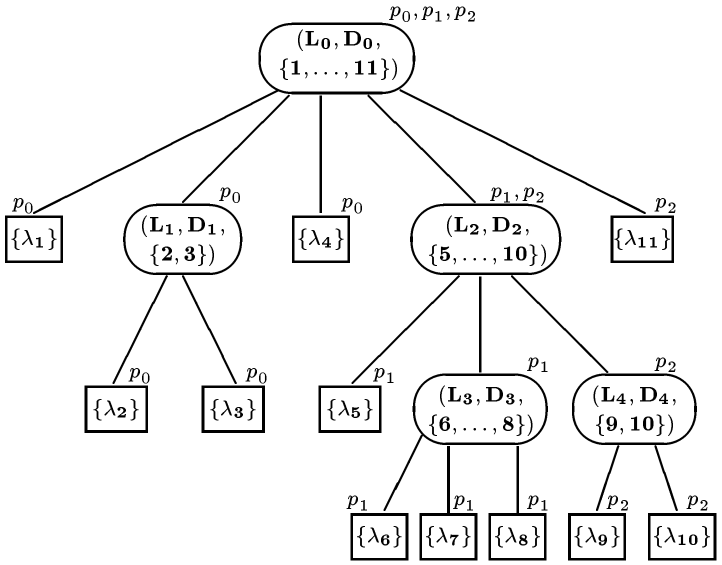

Figure 5: Representation tree annotated to describe the execution of the parallel algorithm. The matrix size is 11; the algorithm is executed by 3 processors.

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 these other two stages that dominate the computation time. In this section, we give a very 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 well-understood, is scalable, and can achieve high performance [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 8000 to 128000 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 guaranteed high relative accuracy [39]. However the dqds algorithm is inherently sequential; for the sake of simplicity in the implementation this computation is performed redundantly on all the

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**	**5.9**
	PDSTEDC	–	–	32.3	19	11.3
Backtransformation	PLAPACK	178	95	51	29	18
	ScaLAPACK	–	–	113	65	34

Table 1: Timings in seconds for different stages of the dense eigensolvers, $n = 8000$, 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**	**20.2**	**19.2**
	PDSTEDC	–	92.4	52.0	32.5
Backtransformation	PLAPACK	292	160	93	65.7
	ScaLAPACK	–	226	114	64.2

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

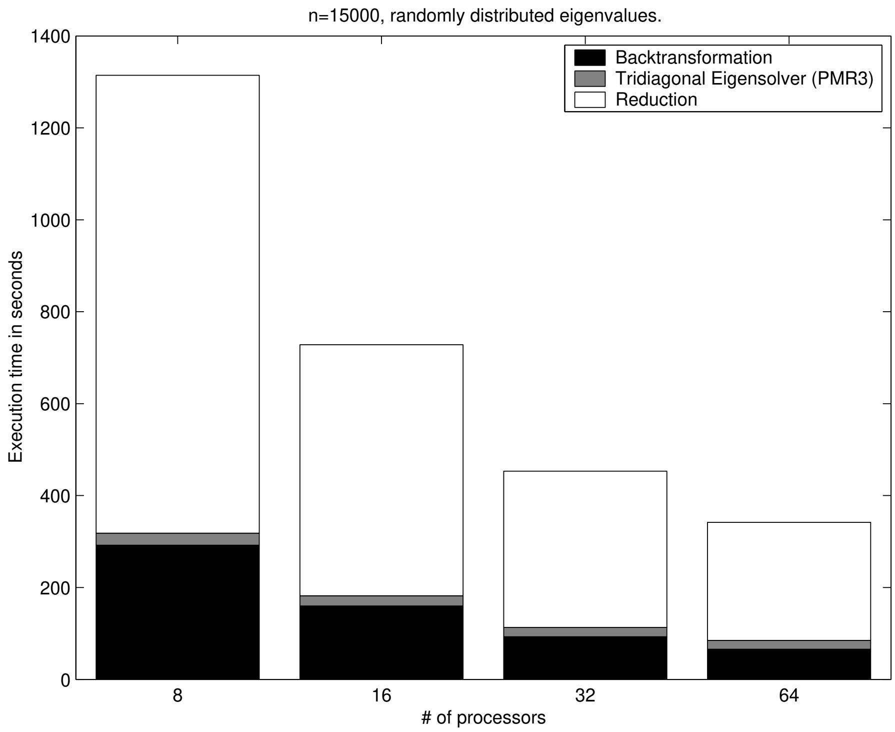

Figure 6: Timing Breakdown for the Dense PMR^3 Algorithm, $n = 15000$.

processors, hence the sluggish speedups for PMR^3 in Table 2. It is possible to obtain good speedups for the initial eigenvalue computation by using bisection with a differential qd transform; however bisection is extremely slow. In order to give an idea, for the computation of all the eigenvalues of a matrix of size 8000 (eigenvalues uniformly distributed on [0,1]), the sequential dqds algorithm runs faster than the ScaLAPACK parallel implementation of bisection (PDSTEBZ) executed on 16 processors. Bisection is easily parallelized, but some care needs to be exercised to guarantee accuracy [20]. Given the small percentage of total time consumed by the tridiagonal problem, as seen in Figure 6, we have chosen not to implement parallel bisection in PLAPACK since its value is marginal.

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 on matrices of sizes 8000, 16000, 32000, 64000 and 128000 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 & 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; it is also evident that within the tridiagonal stage the eigenvalue computation is dominant when the number of processors is more than 16.

n	p	Reduction	PMR^3	Back.	Total (Dense PMR^3)	PLAPACK QR
8000	1	1081 s.	17.6 s. (5.4 s.)	348 s.	**24.1 min.**	100 min.*
16000	4	2425 s.	38.8 s. (20.8 s.)	684 s.	**52.4 min.**	215 min.*
32000	16	4876 s.	118 s. (97 s.)	1388 s.	**106.3 min.**	420 min.*
64000	64	9638 s.	406 s. (386 s.)	2846 s.	**214.8 min.**	750 min.*
128000	256	22922 s.	1510 s. (1490 s.)	5827 s.	**504.3 min.**	

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.

Indicating by $T_p(n)$ the total time from Table 3 to solve a problem of size n with p processors, in Figure 7 we plot *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 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 [28].

[1]The performance of the QR algorithm applied to tridiagonal matrices is very predictable; the extrapolated timings have been obtained running the tridiagonal problem with a larger number of processors (avoiding memory problems) and summing up the adjusted time to the times for reduction and backtransformation.

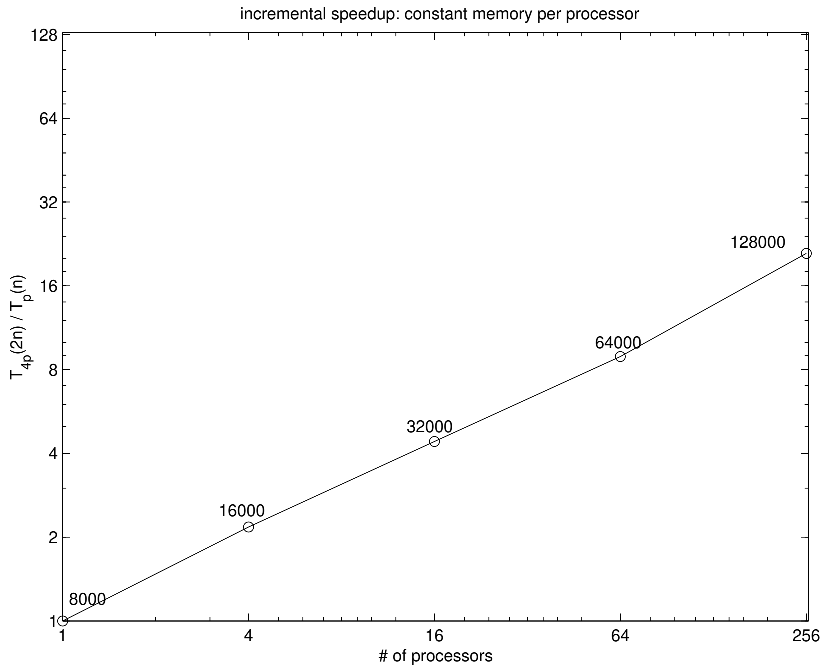

Figure 7: Incremental speedup

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
15000	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

Table 4: Orthogonality and residual results for Dense PMR^3. Given the dense matrix A and with $\hat{\Lambda}$ denoting the computed eigenvalues, and $\hat{V}$ the computed eigenvectors we display here $\max_{i,j} |\hat{V}\hat{V}^T - I|_{ij}$ (orthogonality), and $\max_{i,j} |A\hat{V} - \hat{V}\hat{\Lambda}|_{ij}$ (residual).

4.3 Results for the Tridiagonal Problem

We now focus on the tridiagonal stage, performing comparisons with three different algorithms on matrices with differing eigenvalue distributions. Using earlier algorithms the tridiagonal eigenproblem was 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 for the QR algorithm, 106 seconds for bisection and inverse iteration, 4 seconds for the Divide & 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 & 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, 679 for bisection and inverse iteration, 176 seconds for Divide & Conquer, and 3.4 seconds for MR^3.

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 & 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 noticeably 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`). The symbol $*$ in a few entries of the tables indicates that the experiment has not been run because of excessive time needed.

Distribution	Method	# of processors					
		1	2	4	8	16	32
RANDOM	PMR^3	**17.4**	**13**	**9.4**	**7.4**	**6.2**	**5.9**
	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**	**6.1**
	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

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

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`, and since `PDSTEIN` does not split clusters across processors, all eigenvectors of the cluster end up being computed 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 does not exhibit particularly good speedup due to the redundant serial eigenvalue computation. It is interesting to analyze the breakdown of the timing between the eigenvalue and eigenvector computations; we display these values in Table 6. Notice that once the eigenvalues have been computed the algorithm attains almost perfect speedup. 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 15000 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 and the corresponding speedups are displayed in Figure 8. The algorithm attains good performance and the speedup of the eigenvector computation is almost

Distribution	Eigenvalues	Eigenvectors					
		$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
RANDOM	5.4	12	7.6	4.0	2.0	1.0	.5
UNIFORM	5.6	9.5	6.6	3.4	1.7	.9	.4
GEOMETRIC	3.2	7.8	6.2	3.0	1.6	.8	.4
CLUSTERED	.001	.1	.059	.05	.042	.039	.037

Table 6: Breakdown of timings in seconds for PMR^3, $n = 8000$: Time spent in the computation of eigenvalues (second column) and eigenvectors.

Distribution	Method	# of processors			
		8	16	32	64
RANDOM	PMR^3	**26.2**	**22.2**	**20.2**	**19.2**
	`PDSTEDC`	–	92.4	52.0	32.5
	`PLA_VDVt`	$*$	2054	1000	565
UNIFORM	PMR^3	**26.4**	**22.7**	**20.9**	**20.0**
	`PDSTEDC`	–	101.5	56.8	35.3
	`PLA_VDVt`	$*$	2172	991	564
GEOMETRIC	PMR^3	**18.4**	**15.0**	**13.4**	**12.4**
	`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

Table 7: Timings in seconds for tridiagonal eigensolvers, $n = 15000$.

perfect once the eigenvalues have been computed.

Matrix	Size	Eigenvalues	Eigenvectors					
			$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
Biphenyl	966	.072	.133	.091	.05	.027	.014	.007
$SiOSi_6$	1687	.235	.436	.274	.155	.085	.043	.022
ZSM-5	2053	.339	.752	.445	.228	.122	.066	.033
auto.7923	7923	4.3	11.2	7.3	3.9	2.0	1.0	.52
auto.12387	12387	15.8	31.6	20.0	10.2	5.2	2.6	1.3
auto.13786	13786	13.5	31.8	22.3	11.3	5.7	2.8	1.4

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 finite element modeling of automobile bodies.

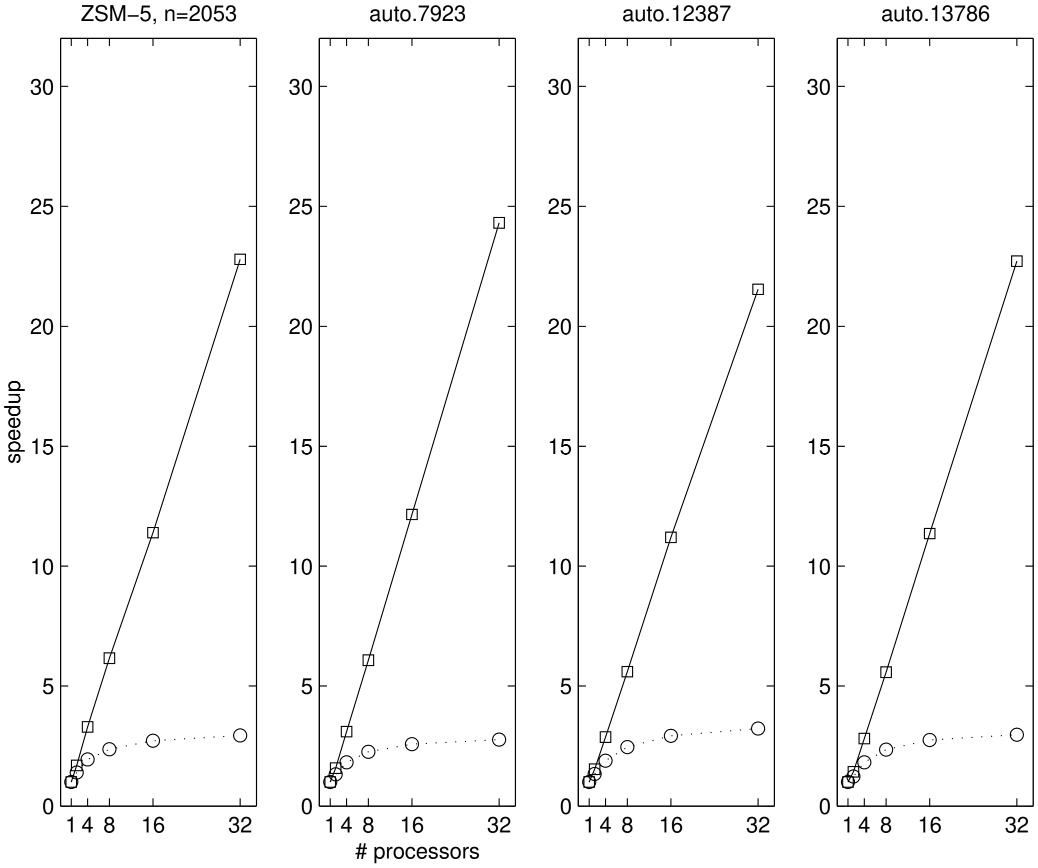

Figure 8: Experiments on matrices arising in applications. The solid line refers to the speedup for the eigenvector computation in PMR^3, while the dotted line is the speedup for the total computation in PMR^3 (including the redundant eigenvalue computation on each processor).

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 the other algorithms, but also scales up well in memory requirements ($O(n)$ work space). This allows for very large eigenproblems to be solved faster than with other methods. In particular, the time now spent in the tridiagonal eigenproblem is negligible 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.

6 Acknowledgments

This research was supported in part by NSF CAREER Award No. ACI-0093404 (ISD) and NSF Award No. ACI-0203685. 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] R. C. Agarwal, F. Gustavson, and M. Zubair. A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. *IBM Journal of Research and Development*, 38(6), 1994.

[2] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*, 1997.

[3] 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 (second edition)*. SIAM, Philadelphia, 1995. 324 pages.

[4] V. Bargmann, D. Montgomery, and J. von Neumann. Solution of linear systems of high order. Report prepared for Navy Bureau of Ordnance, 1946. Reprinted in [71, pp. 421–477].

[5] D. Bernholdt and R. Harrison. Orbital invariant second order many-body perturbation on parallel computers: An approach for large molecules. *J. Chem. Physics.*, 1995.

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

[7] M. Berry and A. Sameh. Multiprocessor Jacobi algorithms for dense symmetric eigenvalue and singular value decompositions. In *Proc. International Conference on Parallel Processing*, pages 433–440, 1986.

[8] Paolo Bientinesi, Enrique S. Quintana-Orí, and Robert A. van de Geijn. Flame@lab: A farewell to indices. Technical Report FLAME Working Note 11, CS-TR-03-11, Department of Computer Sciences, The University of Texas at Austin, April 2003.

[9] 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 *Scalable High Performance Computing Conference*, Knoxville, TN, May 1994.

[10] Christian Bischof and Charles Van Loan. The WY representation for products of Householder matrices. *SIAM J. Sci. Stat. Comput.*, 8(1):s2–s13, Jan. 1987.

[11] 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*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[12] R. Brent and F. Luk. The solution of singular value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Stat. Comput.*, 6:69–84, 1985.

[13] R. P. Brent. *Algorithms for minimization without derivatives.* Prentice-Hall, 1973.

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

[15] Almadena Chtchelkanova, John Gunnels, Greg Morrow, James Overfelt, and Robert A. van de Geijn. Parallel implementation of BLAS: General techniques for level 3 BLAS. *Concurrency: Practice and Experience*, 9(9):837–857, September 1997.

[16] M. Chu. A simple application of the homotopy method to symmetric eigenvalue problems. *Lin. Alg. Appl*, 59:85–90, 1984.

[17] M. Chu. A note on the homotopy method for linear algebraic eigenvalue problems. *Lin. Alg. Appl*, 105:225–236, 1988.

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

[19] T. J. Dekker. Finding a zero by means of successive linear interpolation. In B. Dejon and P. Henrici, editors, *Constructive aspects of the fundamanetal theorem of algebra.* New York: Wiley-Interscience, 1969.

[20] J. Demmel, I. Dhillon, and H. Ren. On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic. *Electronic Trans. Num. Anal.*, 3:116–140, December 1995. LAPACK working note 70.

[21] J. Demmel and K. Stanley. The performance of finding eigenvalues and eigenvectors of dense symmetric matrices on distributed memory computers. In *Proceedings of the Seventh SIAM Conference on Parallel Proceesing for Scientific Computing*. SIAM, 1994.

[22] J. Demmel and K. Veselić. Jacobi's method is more accurate than QR. *SIAM J. Mat. Anal. Appl.*, 13(4):1204–1246, 1992. (also LAPACK Working Note #15).

[23] J. W. Demmel. *Applied Numerical Linear Algebra.* SIAM, Philadelphia, PA, 1997.

[24] I. S. Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem.* PhD thesis, Computer Science Division, University of California, Berkeley, California, May 1997. Available as UC Berkeley Technical Report No. UCB//CSD-97-971.

[25] I. S. Dhillon. Current inverse iteration software can fail. *BIT*, 38(4):685–704, December 1998.

[26] 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, March 1997. SIAM.

[27] I. S. Dhillon and A. N. Malyshev. Inner deflation for symmetric tridiagonal matrices. *Lin. Alg. Appl.*, 358:139–144, October 2002.

[28] I. S. Dhillon and B. N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Lin. Alg. Appl.*, 2003. To Appear.

[29] I. S. Dhillon and B.N. Parlett. Orthogonal Eigenvectors and Relative Gaps. *SIAM J. Matrix Anal. Appl.*, 2003. To Appear. Also LAPACK Working Note 154 (ut-cs-02-474).

[30] J. Dongarra, J. Du Croz, I. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[31] J. Dongarra, J. Du Croz, S. Hammarling, and Richard J. Hanson. An Extended Set of FORTRAN Basic Linear Algebra Subroutines. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[32] J. Dongarra, R. Hempel, A. Hay, and D. Walker. A proposal for a user-level message passing interface in a distributed memory environment. Technical Report ORNL/TM-12231, Oak Ridge National Laboratory, Oak Ridge, TN, February 1993.

[33] J. Dongarra and D. Sorensen. A fully parallel algorithm for the symmetric eigenproblem. *SIAM J. Sci. Stat. Comput.*, 8(2):139–154, March 1987.

[34] J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable dense linear algebra libraries. In *Scalable High-Performance Computing Conference*. IEEE Computer Society Press, April 1992.

[35] Jack J. Dongarra, Sven J. Hammarling, and Danny C. Sorensen. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27, 1989.

[36] C. Edwards, P. Geng, A. Patra, and R. van de Geijn. Parallel matrix distributions: have we been doing it all wrong? Technical Report TR-95-40, Department of Computer Sciences, The University of Texas at Austin, 1995.

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

[38] G.I. Fann, R. J. Littlefield, and D.M. Elmwood. Performance of a fully parallel dense real symmetric eigensolver in quantum chemistry appliacation. In *Proceedings of the High Performance Computing '95, Simulation MultiConference*. The Society of Computer Simulation, San Diego, 1995.

[39] K. Fernando and B. Parlett. Accurate singular values and differential qd algorithms. *Numerische Mathematik*, 67(2):191–229, 1994.

[40] K. V. Fernando. On computing an eigenvector of a tridiagonal matrix. Part I: Basic results. *SIAM J. Matrix Anal. Appl.*, 18(4):1013–1034, October 1997.

[41] G. J. F. Francis. The QR transformation, parts I and II. *Computer J.*, 4:265–271,332–345, 1961-62.

[42] Wallace J. Givens. Numerical computation of the characteristic values of a real symmetric matrix. Technical Report ORNL-1574, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 1954.

[43] G. H. Golub. Some modified matrix eigenvalue problems. *SIAM Review*, 15:318–334, 1973.

[44] Kazushige Goto and Robert A. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-55, Department of Computer Sciences, The University of Texas at Austin, 2002.

[45] M. Gu and S. C. Eisenstat. A stable and efficient algorithm for the rank-1 modification of the symmetric eigenproblem. *SIAM J. Mat. Anal. Appl.*, 15(4):1266–1276, October 1994. Yale Tech report YALEU/DCS/RR-916, Sept 1992.

[46] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM J. Mat. Anal. Appl.*, 16(1):172–191, January 1995.

[47] John Gunnels, Calvin Lin, Greg Morrow, and Robert 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 IPPS/SPDP '98)*, pages 110–116, 1998.

[48] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[49] John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.

[50] B. Hendrickson, E. Jessup, and C. Smith. Toward an efficient parallel eigensolver for dense symmetric matrices. *SIAM J. Sci. Comput.*, 20(3):1132–1154, 1999.

[51] B. Hendrickson, R. Leland, and S. Plimpton. A parallel algorithm for matrix-vector multiplication. Technical report, Sandia National Laboratories, 1993.

[52] B. A. Hendrickson and D. E. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. 15(5):1201–1226, 1994.

[53] C. G. F. Jacobi. Concerning an easy process for solving equations occurring in the theory of secular disturbances. *J. Reine Angnew. Math.*, 30:51–94, 1846.

[54] E. Jessup and I. Ipsen. Improving the accuracy of inverse iteration. *SIAM J. Sci. Stat. Comput.*, 13(2):550–572, 1992.

[55] W. Kahan. Notes on Laguerre's iteration. University of California Computer Science Division preprint, 1992.

[56] M. F. Kaplan. *Implementation of Automated Multilevel Substructuring for Frequency Response Analysis of Structures.* PhD thesis, University of Texas, Austin, Texas, 2001.

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

[58] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[59] J. G. Lewis and R. A. van de Geijn. Implementing matrix-vector multiplication and conjugate gradient algorithms on distributed memory multicomputers. In *Proceedings of Supercomputing 1993*, 1993.

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

[61] T. Y. Li and Z. Zeng. The Laguerre iteration in solving the symmetric tridiagonal eigenproblem, revisited. *SIAM J. Sci. Stat. Comput.*, 15(5):1145–1173, September 1994.

[62] T.-Y. Li, H. Zhang, and X. H. Sun. Parallel homotopy algorithm for symmetric tridiagonal eigenvalue problems. *SIAM J. Sci. Stat. Comput.*, 12:464–485, 1991.

[63] B. Parlett. *The Symmetric Eigenvalue Problem.* SIAM, Philadelphia, PA, second edition, 1997.

[64] B. N. Parlett. Laguerre's Method Applied to the Matrix Eigenvalue Problem. *Math. Comp.*, 18:464–485, 1964.

[65] B. N. Parlett and I. S. Dhillon. Fernando's solution to Wilkinson's problem: An application of double factorization. *Lin. Alg. Appl.*, 267:247–279, December 1997.

[66] B. N. Parlett and I. S. Dhillon. Relatively robust representations of symmetric tridiagonals. *Lin. Alg. Appl.*, 309:121–151, 2000.

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

[68] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Transactions on Mathematical Software*, 29(2):218–243, June 2003.

[69] Christopher Smith, Bruce Hendrickson, and Elizabeth Jessup. A parallel algorithm for householder tridiagonalization.

[70] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

[71] A. H. Taub, editor. *John von Neumann Collected Works*, volume V, Design of Computers, Theory of Automata and Numerical Analysis. Pergamon, Oxford, 1963.

[72] 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(6):2223–2226, 1999.

[73] R. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.

[74] R.A. van de Geijn. Storage schemes for parallel eigenvalue algorithms. In G.H. Golub and P. van Dooren, editors, *Numerical Linear Algebra, Digital Signal Processing and Parallel Algorithms*, pages 639–648. NATO ASI Series, Springer Verlag, 1991.

[75] Robert van de Geijn and Jerrell Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, April 1997.

[76] Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME API. *ACM Trans. Math. Soft.* submitted.

[77] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98*, 1998.

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

Appendix

Please note that we have supplied this appendix for the sake of completeness. We do NOT plan to include this appendix in the eventual publication. References in the text to the appendix will be replaced by references to the technical report.

A Householder transformations (reflectors)

Given vector x, partition

$$x = \left(\frac{\chi_1}{x_2} \right)$$

where χ_1 equals the first element of x.

Define the Householder vector associated with x as the vector

$$u = \left(\frac{1}{x_2/\nu_1} \right) = \left(\frac{1}{\hat{u}} \right),$$

where $\nu_1 = \chi_1 + \text{sign}(\chi_1)\|x\|_2$, and let $\beta = \frac{2}{u^T u}$. Then $(I - \beta u u^T)x = \eta \|x\|_2 e_1$ where η equals the scalar $-\text{sign}(\chi_1)\|x\|_2$. The transformation $I - \beta u u^T$, with $\beta = 2/u^T u$ is referred to as a Householder transformation or reflector.

Let us introduce the notation $[\eta, \hat{u}, \beta] := h(x)$ as the computation of the above mentioned η, $\hat{u}$, and β from given vector x. An important feature of $I - \beta u u^T$ is that it is orthonormal

$$(I - \beta u u^T)^T (I - \beta u u^T) = (I - \beta u u^T)(I - \beta u u^T)^T = I$$

and symmetric, $(I - \beta u u^T)^T = I - \beta u u^T$.

B Reduction to tridiagonal form

Given an $n \times n$ symmetric matrix A, we now discuss how to reduce A to a tridiagonal matrix using unitary similarity transformations, which have the property that they preserve eigenvalues. More precisely, we wish to compute Householder transforms $\{H_1, \ldots, H_{n-2}\}$ such that

$$T = H_{n-2} \cdots H_1 A H_1 \cdots H_{n-2}$$

is a symmetric tridiagonal.

B.1 An unblocked algorithm

Given a matrix A, partition it like

$$A \rightarrow \left(\begin{array}{c|c} \alpha_{11} & a_{21}^T \\ \hline a_{21} & A_{22} \end{array} \right)$$

Algorithm: $[U\backslash T, s] = \textsc{TriRedUnblocked}(A)$

Partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$, $b \rightarrow \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right)$

where A_{TL} is 0×0 and s_T has 0 elements

while $size(A_{BR}) \neq 0 \times 0$ **do**

Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & a_{01} & A_{02} \\ \hline\hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right) \rightarrow \left(\begin{array}{c} s_0 \\ \hline\hline \beta_1 \\ \hline s_2 \end{array}\right)$$

where α_{11} and β_1 are scalars

Compute Householder vector

$[\hat{u}, \nu_1, \beta_1] = h(a_{21})$

$u = \left(\begin{array}{c} 1 \\ \hline \hat{u} \end{array}\right)$

Store ν_1 **and** $\hat{u}$

$a_{21} := \left(\begin{array}{c} \nu_1 \\ \hline \hat{u} \end{array}\right)$

Update A_{22}

$z := \beta_1 A_{22} u$

$w := z - \frac{\beta_1}{2}(z^T u)u$

$A_{22} := A_{22} - uw^T - wu^T$

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right) \leftarrow \left(\begin{array}{c} s_0 \\ \hline \beta_1 \\ \hline\hline s_2 \end{array}\right)$$

enddo

Figure 9: Unblocked reduction to tridiagonal form.

where α_{11} is a scalar. Now, if one lets $[\eta, \hat{u}, \beta_1] = h(a_{21})$, then

$$\begin{aligned}
&\left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & I - \beta_1 uu^T \end{array}\right) \left(\begin{array}{c|c} \alpha_{11} & a_{21}^T \\ \hline a_{21} & A_{22} \end{array}\right) \left(\begin{array}{c|c} 1 & 0 \\ \hline 0 & I - \beta_1 uu^T \end{array}\right) \\
&= \left(\begin{array}{c|c} \alpha_{11} & \left(\begin{array}{c|c} \eta & 0 \end{array}\right) \\ \hline \left(\begin{array}{c} \eta \\ \hline 0 \end{array}\right) & (I - \beta_1 uu^T) A_{22} (I - \beta_1 uu^T) \end{array}\right) \\
&= \left(\begin{array}{c|c} \alpha_{11} & \left(\begin{array}{c|c} \eta & 0 \end{array}\right) \\ \hline \left(\begin{array}{c} \eta \\ \hline 0 \end{array}\right) & A_{22} - \beta_1 uu^T A_{22} - \beta_1 A_{22} uu^T + \beta_1^2 u^T A_{22} uuu^T \end{array}\right) \\
&= \left(\begin{array}{c|c} \alpha_{11} & \left(\begin{array}{c|c} \eta & 0 \end{array}\right) \\ \hline \left(\begin{array}{c} \eta \\ \hline 0 \end{array}\right) & A_{22} - uz^T - zu^T + \beta_1 z^T uuu^T \end{array}\right) \\
&= \left(\begin{array}{c|c} \alpha_{11} & \left(\begin{array}{c|c} \eta & 0 \end{array}\right) \\ \hline \left(\begin{array}{c} \eta \\ \hline 0 \end{array}\right) & A_{22} - uw^T - wu^T \end{array}\right)
\end{aligned}$$

where $z = \beta_1 A_{22} u$ and $w = z - \frac{\beta_1}{2} z^T uu$. Notice that $\hat{u}$ can be stored over the zeroes thus introduced in the matrix. To compute the complete reduction to tridiagonal form, one simply continues similarly with the updated matrix A_{22}. An algorithm is given in Fig. 9.

Algorithm: $[A_BL, W, A_{TR}, s] = \text{AccumulateTriRed}(A, b)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $s \rightarrow \left(\begin{array}{c} s_T \\ \hline s_B \end{array}\right)$

Create matrix W of width b: $W \rightarrow \left(\begin{array}{c|c} W_{TL} & W_{TR} \\ \hline W_{BL} & W_{BR} \end{array}\right)$

where A_{TL} and W_{TL} are $0 \times 0$and s_T has 0 elements

while $size(A_{TL}) \neq b \times b$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} s_T \\ \hline s_B \end{array}\right) \rightarrow \left(\begin{array}{c} s_0 \\ \hline \beta_1 \\ \hline s_2 \end{array}\right), \text{ and}$$

$$\left(\begin{array}{c|c} W_{TL} & W_{TR} \\ \hline W_{BL} & W_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} W_{00} & w_{01} & W_{02} \\ \hline w_{10}^T & \omega_{11} & w_{12}^T \\ \hline W_{20} & w_{21} & W_{22} \end{array}\right)$$

where α_{11}, β_1, and ω_{11} are scalar

Update α_{11} **and** a_{21}

$$\left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right) := \left(\begin{array}{c} \alpha_{11} \\ \hline a_{21} \end{array}\right) - \left(\begin{array}{c} a_{10}^T \\ \hline A_{20} \end{array}\right) w_{10} - \left(\begin{array}{c} w_{10}^T \\ \hline W_{20} \end{array}\right) a_{10}^T$$

Compute Householder vector

$[\hat{u}, \nu_1, \beta_1] = h(a_{21})$

$u = \left(\begin{array}{c} 1 \\ \hline \hat{u} \end{array}\right)$

Store ν_1 **and** $\hat{u}$

$a_{21} := \left(\begin{array}{c} \nu_1 \\ \hline \hat{u} \end{array}\right)$

Update A_{22}

$z := \beta_1(A_{22}u - A_{20}W_2^T u - W_2 A_{20}^T u)$

$w_{21} := z - \frac{\beta_1}{2}(z^T u)u$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} s_T \\ \hline s_B \end{array}\right) \leftarrow \left(\begin{array}{c} s_0 \\ \hline \beta_1 \\ \hline s_2 \end{array}\right), \text{ and}$$

$$\left(\begin{array}{c|c} W_{TL} & W_{TR} \\ \hline W_{BL} & W_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} W_{00} & w_{01} & W_{02} \\ \hline w_{10}^T & \omega_{11} & w_{12}^T \\ \hline W_{20} & w_{21} & W_{22} \end{array}\right)$$

enddo

Figure 10: Accumulate U and W

B.2 A blocked algorithm

It is well-known that by casting an algorithm in terms of matrix-matrix operations (level 3 BLAS), superior performance can be expected [30]. We now show how part of the computation can be cast in terms of matrix-matrix multiplication.

Consider the update

$$A := A - uw^T - wu^T.$$

If it were possible to accumulate updates $\{u_1, \ldots, u_k\}$ and $\{w_1, \ldots, w_k\}$, from a number of iterations, we would update

$$A := A - u_1 w_1^T - w_1 u_1^T. - \cdots - u_k w_k^T - w_k u_k^T = A - UW^T - WU^T$$

where the jth columns of U_k and W_k equal u_j and w_j, respectively. Now this computation can be cast in terms of a special matrix-matrix multiplication that only updates the lower triangular part of A_{22} called a *symmetric rank-2k update.*

Algorithm: $[U\backslash T, s] = \textsc{TriRedBlocked}(A)$

Partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$, $b \rightarrow \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right)$

where A_{TL} is 0×0 and s_T has 0 elements

while $size(A_{BR}) \neq 0 \times 0$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right) \rightarrow \left(\begin{array}{c} s_0 \\ \hline\hline s_1 \\ \hline s_2 \end{array}\right)$$

where A_{11} is $b \times b$ and s_1 has b elements

Compute $[U_2, W_2, \{U\backslash H\}_1, s_1] = \text{AccumulateTriRed}(A_{BR}, b)$

Store $\{U\backslash H\}_1$ **and** U_2

$A_{11} := \{U\backslash H\}_1$

$A_{21} := U_2$

Update A_{22}

$A_{22} := A_{22} - U_2 W_2^T - W_2 U_2^T$

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & a_{21} & A_{22} \end{array}\right) \text{ and } \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right) \leftarrow \left(\begin{array}{c} s_0 \\ \hline s_1 \\ \hline\hline s_2 \end{array}\right)$$

enddo

Figure 11: Blocked reduction to tridiagonal form.

The problem is that to compute u_j and w_j, the part of the jth column of A, updated with $(j-1)$ transformations, below the diagonal must be available. If we delay updating part of the matrix, then that column is not available. However, notice that the j column of the updated A equals

$$A_j e_j = (A - U_{j-1} W_{j-1}^T - W_{j-1} U_{j-1}^T) e_j = A e_j - U_{j-1} W_{j-1}^T e_j - W_{j-1} U_{j-1}^T e_j.$$

This then gives us the information necessary to compute u_j and β_j. Also,

$$z_j = \beta_j A_j u_j = \beta_j (A u_j - U_{j-1} W_{j-1}^T u_j - W_{j-1} U_{j-1}^T u_j)$$

which allows the computation of $w_j = z_j - \frac{\beta_j}{2} z_j^T u_j u_j$. A few of the details are missing in the above argument, which are filled in in the algorithm in Fig. 10. Once a block of updates have been computed, they are applied to the appropriate submatrix, as shown in the blocked algorithm in Fig. 11.

Note 1 *It is important to realize that the blocked algorithm in Fig. 11 only casts roughly half the computation in terms of symmetric rank-2k updates. The other half is in the symmetric matrix-vector multiplication $A_{22} a_{21}$. This is inherent in the problem. Thus, only half the computation can be sped-up by exploiting matrix-matrix operations. Essentially all of the other half is inherently in so-called matrix-vector (level-2 BLAS) operations [31]. Thus, regardless of how fast the symmetric rank-2k update is, the best one can hope for is a factor two improvement in performance relative to the unblocked algorithm.*

B.3 Implementation

As part of the Formal Linear Algebra Methods Environment (FLAME) project, we have developed a number of APIs that the algorithms given in this appendix to be represented in code so that the code closely mirrors the algorithms [48, 76, 68]. In Figures 15–17, we show the algorithms as coded using the FLAME/Matlab (FLAME@lab) interface [8]. The parallel implementations of the Reduction and Backtransformation are coded using an extension of the PLAPACK API that allows the parallel implementation to look much like the FLAME@lab code. It is interesting to note that the derivation, implementation using FLAME@lab, and finally the implementation with PLAPACK took less than a day, including debugging. Rather than discussing this parallel implementation in detail, we restrict ourselves to a few high-level insights.

It is well-known that dense linear algebra operations like the reduction to tridiagonal form require a so-called two-dimensional block-cyclic distribution for scalability and load balance [69, 36, 52, 70]. The p processors are viewed as a logical $r \times c$ mesh, with $p = rc$. Assume a matrix C is partitioned like

$$C = \left(\begin{array}{c|c|c|c} \hat{C}_{00} & \hat{C}_{01} & \cdots & \hat{C}_{0(N-1)} \\ \hline \hat{C}_{10} & \hat{C}_{11} & \cdots & \hat{C}_{1(N-1)} \\ \hline \vdots & \vdots & & \vdots \\ \hline \hat{C}_{(N-1)0} & \hat{C}_{(N-1)1} & \cdots & \hat{C}_{(N-1)(N-1)} \end{array} \right)$$

where C_{ij} is $n_b \times n_b$. In its simplest form, block cyclic distribution will assign C_{ij} to processor $(i\%r, j\%c)$, where % represents the "mod" operation. This is the distribution used by ScaLAPACK. PLAPACK, which was used for our implementations, uses a block-cyclic distribution that is a variation on this approach.

Now, the parallel implementation of the main two computations, $z := A_{22}u$ (symmetric matrix-vector multiplication) and $A_{22} := A_{22} - U_2W_2^T - W_2U_2^T$ (symmetric rank-2k update), is well-understood [51, 59, 15, 73]. Both are supported as library routines by ScaLAPACK and PLAPACK. The remaining operations (in the accumulation of U and W) represent a lower-order, though significant, cost. Here there are two different approaches that have been implemented in practice:

- The algorithmic block size b is chosen so that $\left(\frac{A_{11}}{A_{21}} \right)$ resides within a single column of processors. The advantage of this approach is that most computation in the accumulation of U and W is shared among only r processors, which reduces communication overhead for smaller matrices. The disadvantage is that it restricts the algorithmic block size, b, to be bounded by the distribution block size, n_b. This forces a large distribution block size, which hampers load balance, or a small algorithmic block size, which hampers the performance of the symmetric rank-2k update. Also, for very large problems, it leaves most processors idle during these computations. This approach is implemented by ScaLAPACK.

- The current panel $\left(\frac{A_{11}}{A_{21}} \right)$ is redistributed to the processors so that it is partitioned by row-blocks of size n_b. The advantage of this approach is that most computation in the accumulation of U and W is shared among p processors. Also, now n_b and b are not coupled. The disadvantage is that it increases communication overhead for smaller matrices, since more processors are involved in communication. It is this approach that we chose to implement for this research, using PLAPACK.

It should be noted that while the first approach can be implemented using PLAPACK, the second approach can inherently not be implemented by ScaLAPACK.

Algorithm: $\mathrm{QxBUnblocked}(A, s, B)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $s \rightarrow \left(\begin{array}{c} s_T \\ \hline s_B \end{array}\right)$, and $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$

where A_{BR} is 1×1, s_B has 1 element, and B_B has 1 row

while $size(A_{TL}) \neq 0 \times 0$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} s_T \\ \hline s_B \end{array}\right) \rightarrow \left(\begin{array}{c} s_0 \\ \hline \beta_1 \\ \hline s_2 \end{array}\right), \text{ and } \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline B_2 \end{array}\right)$$

where α_{11} and β_1 are scalars, and b_1^T is a row

Retrieve Householder vector from A

Partition $a_{21} \rightarrow \left(\begin{array}{c} \alpha_{21T} \\ \hline a_{21B} \end{array}\right)$ where α_{21T} is a scalar

$u = \left(\begin{array}{c} 1 \\ \hline a_{21B} \end{array}\right)$

Update B_2

$z := \beta_1 u^T B_2$

$B_2 := B_2 - u z^T$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} s_T \\ \hline s_B \end{array}\right) \leftarrow \left(\begin{array}{c} s_0 \\ \hline \beta_1 \\ \hline s_2 \end{array}\right), \text{ and } \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline b_1^T \\ \hline B_2 \end{array}\right)$$

enddo

Figure 12: Unblocked backtransformation.

C Backtransformation

Notice that the reduction to tridiagonal form computes a matrix Q such that $Q^T A Q = T$. The tridiagonal eigensolver computes the eigenvectors of T as the columns of matrix B. Now, the eigenvectors of A are given by the columns of matrix QB. Note, however, that the reduction step leaves Q in factored form, as the Householder vectors that were encountered. Thus, we need to form

$$QB = (I - \beta_1 u_1 u_1^T)(I - \beta_2 u_2 u_2^T) \cdots (I - \beta_{n-2} u_{n-2} u_{n-2}^T) B$$

C.1 An unblocked algorithm

Noting that the Householder vectors are stored under the first subdiagonal of the updated A and the "1"s are implicitly given, an unblocked algorithm for applying the backtransformation is given in Fig. 12.

C.2 A blocked algorithm

To cast the computation mostly in terms of matrix-matrix multiplications, it can be recognized that given k Householder vectors $\{u_1, \ldots, u_k\}$ and the corresponding "β"s, there exists a lower triangular matrix L_k so that

$$(I - \beta_k u_k u_k^T) \cdots (I - \beta_2 u_2 u_2^T)(I - \beta_1 u_1 u_1^T) = I - U_k L_k U_k^T,$$

Algorithm: $\textsc{ComputeBlkHous}(U, s, L)$

Partition $U \rightarrow \left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline U_{BL} & U_{BR} \end{array}\right)$, $s \rightarrow \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right)$, and $L \rightarrow \left(\begin{array}{c||c} L_{TL} & L_{TR} \\ \hline\hline L_{BL} & L_{BR} \end{array}\right)$

where U_{TL} and L_{TL} are 0×0, and s_T has 0 elements

while $n(U_{BR}) = 0$ **do**

Repartition

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline U_{BL} & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} U_{00} & u_{01} & U_{02} \\ \hline\hline u_{10}^T & v_{11} & u_{12}^T \\ \hline U_{20} & u_{21} & U_{22} \end{array}\right), \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right) \rightarrow \left(\begin{array}{c} s_0 \\ \hline\hline \beta_1 \\ \hline s_2 \end{array}\right), \text{ and}$$

$$\left(\begin{array}{c||c} L_{TL} & L_{TR} \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c||c|c} L_{00} & l_{01} & L_{02} \\ \hline\hline l_{10}^T & \lambda_{11} & l_{12}^T \\ \hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

where v_{11}, β_1, and λ_{11} are scalars

Retrieve Householder vector from U

Partition $u_{21} \rightarrow \left(\begin{array}{c} v_{21T} \\ \hline u_{21B} \end{array}\right)$ where v_{21T} is a scalar

$u = \left(\begin{array}{c} 1 \\ \hline u_{21B} \end{array}\right)$

Update L

$l_{21} := \beta_1 L_{00}^T U_{20}^T u$

$\lambda_{11} := \beta_1$

Continue with

$$\left(\begin{array}{c||c} U_{TL} & U_{TR} \\ \hline\hline U_{BL} & U_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} U_{00} & u_{01} & U_{02} \\ \hline u_{10}^T & v_{11} & u_{12}^T \\ \hline\hline U_{20} & u_{21} & U_{22} \end{array}\right), \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right) \leftarrow \left(\begin{array}{c} s_0 \\ \hline \beta_1 \\ \hline\hline s_2 \end{array}\right), \text{ and}$$

$$\left(\begin{array}{c||c} L_{TL} & L_{TR} \\ \hline\hline L_{BL} & L_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c||c} L_{00} & l_{01} & L_{02} \\ \hline l_{10}^T & \lambda_{11} & l_{12}^T \\ \hline\hline L_{20} & l_{21} & L_{22} \end{array}\right)$$

enddo

Figure 13: Compute L

where the jth column of U_k equals u_j and L_k is $k \times k$. It can be easily shown that

$$L_{k+1} = \left(\begin{array}{c|c} L_k & 0 \\ \hline l_{k+1}^T & \beta_{k+1} \end{array}\right)$$

where $l_{k+1} = \beta_{k+1} L_k^T U_k^T u_{k+1}$. Taking into account the special structure of the Householder vectors, an algorithm for computing matrix L_k is given in Fig. 13.

Now, computing QB can be cast in terms of such block Householder transformations as illustrated in Fig. 14. Notice that for the formation of QB, essentially all of the computation can be cast in terms of matrix-matrix multiplication.

C.3 Implementation

Sequential and parallel implementation of the Backtransformation were again derived using FLAME@lab and PLAPACK. For the parallel implementation, we again assume the same two-dimensional block-cyclic distribution as for the reduction to tridiagonal form.

Again, the parallel implementation of the main computations (both matrix-matrix multiplica-

Algorithm: $\textsc{QxBBlocked}(A, s, B)$

Partition $A \rightarrow \left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right)$, $s \rightarrow \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right)$, and $B \rightarrow \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right)$

where A_{BR} is 1×1, s_B has 1 element, and B_B has 1 row

while $size(A_{TL}) \neq 0 \times 0$ **do**

Determine block size b

Repartition

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c||c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline\hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right) \rightarrow \left(\begin{array}{c} s_0 \\ \hline s_1 \\ \hline\hline s_2 \end{array}\right), \text{ and } \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline\hline B_2 \end{array}\right)$$

where A_{11} is $b \times b$, s_1 has b elements, and B_1 has b rows

Retrieve Householder vectors from A

$U_{11} = unit\_strictly\_lower\_triangular(A_{11})$

$U_{21} = A_{21}$

Create matrix L of same size as A_{11}

$\textsc{ComputeBlkHouse}\left(\left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right), s_1, L\right)$

Update B_1 **and** B_2

$Z := L^T \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right)^T \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right)$

$\left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) := \left(\begin{array}{c} B_1 \\ \hline B_2 \end{array}\right) - \left(\begin{array}{c} U_{11} \\ \hline U_{21} \end{array}\right) Z$

Continue with

$$\left(\begin{array}{c||c} A_{TL} & A_{TR} \\ \hline\hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c||c|c} A_{00} & A_{01} & A_{02} \\ \hline\hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \left(\begin{array}{c} s_T \\ \hline\hline s_B \end{array}\right) \leftarrow \left(\begin{array}{c} s_0 \\ \hline\hline s_1 \\ \hline s_2 \end{array}\right), \text{ and } \left(\begin{array}{c} B_T \\ \hline\hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline\hline B_1 \\ \hline B_2 \end{array}\right)$$

enddo

Figure 14: Blocked computation of QB.

tions),

$$
\begin{aligned}
Z &:= L^T \left(\frac{U_{11}}{U_{21}}\right)^T \left(\frac{B_1}{B_2}\right) \\
\left(\frac{B_1}{B_2}\right) &:= \left(\frac{B_1}{B_2}\right) - \left(\frac{U_{11}}{U_{21}}\right) Z,
\end{aligned}
$$

is well-understood [1, 75, 15, 47]. Both are supported as library routines by ScaLAPACK and PLAPACK.

The computation of L represents a lower-order cost. Here there are two different approaches that have implemented in practice for the computation of L:

- The algorithmic block size b is chosen so that $\left(\frac{A_{11}}{A_{21}}\right)$ resides within a single column of processors. The advantage of this approach is that most computation in the accumulation of L is shared among only r processors, which reduces communication overhead for smaller matrices. The disadvantage is that it restricts the algorithmic block size, b, to be bounded by the distribution block size, n_b. This forces a large distribution block size, which hampers load balance, or a small algorithmic block size, which hampers the performance of the matrix-matrix multiplications. Also, for very large problems, it leaves most processors idle during these computations. This approach is implemented by ScaLAPACK.

- The current panel $\left(\frac{A_{11}}{A_{21}}\right)$ is redistributed to the processors so that it is partitioned by row-blocks of size n_b. The advantage of this approach is that most computation in the accumulation of L is shared among p processors. Also, now n_b and b are not coupled. The disadvantage is that it increases communication overhead for smaller matrices, since more processors are involved in communication. It is this approach that we chose to implement for this research, using PLAPACK.

It should be noted that while the first approach can be implemented using PLAPACK, the second approach can inherently not be implemented by ScaLAPACK.

```
function [ A_out, s_out ] = TriRedUnb( A, s )
%
  [ ATL, ATR,...
    ABL, ABR     ] = FLA_Part_2x2( A, 0, 0, 'FLA_TL' );
  [ sT, ...
    sB ] = FLA_Part_2x1( s, 0, 'FLA_TOP' );

  while( size( ATL ) < size( A )-1 )
    [ A00,  a01,      A02,...
      a10t, alpha11,  a12t,...
      A20,  a21,      A22      ] = FLA_Repart_2x2_to_3x3( ATL, ATR,...
                                                            ABL, ABR,...
                                                            1, 1, 'FLA_BR' );
    [ s0, ...
      beta1, ...
         s2 ] = FLA_Repart_2x1_to_3x1( sT, ...
                                  sB, 1, 'FLA_BOTTOM' );

%* ******************************************************************** *%

  [ eta1, uhat, beta1 ] = HouseTrans( a21 );
  u = [ 1
        uhat ];
  a21 = [ eta1
          uhat ];
  z = beta1 * symml( A22 ) * u ;
  w = z - beta1/2 * (z' * u) * u;
  A22 = A22 - tril( u * w' + w * u' );

%* ******************************************************************** *%
    [ ATL, ATR,...
      ABL, ABR ] = FLA_Cont_with_3x3_to_2x2( A00,  a01,      A02,...
                                             a10t, alpha11, a12t,...
                                             A20,  a21,      A22, 'FLA_TL' );
    [ sT, ...
      sB ] = FLA_Cont_with_3x1_to_2x1( s0, ...
                                       beta1, ...
                                       s2, 'FLA_TOP' );
  end

  A_out = [ ATL, ATR
            ABL, ABR ];
  s_out = [ sT
            sB ];

  return;
```

Figure 15: FLAME@lab code for unblocked reduction to tridiagonal form algorithm.

```
function [ A_out, W_out, s_out ] = AccumTriRed( A, W, s )

  [ ATL, ATR,...
    ABL, ABR     ] = FLA_Part_2x2( A, 0, 0, 'FLA_TL' );
  [ WTL, WTR,...
    WBL, WBR     ] = FLA_Part_2x2( W, 0, 0, 'FLA_TL' );
  [ sT, ...
    sB ] = FLA_Part_2x1( s, 0, 'FLA_TOP' );

while( size( WBR, 2 ) ~= 0 )
    [ A00,  a01,      A02,...
      a10t, alpha11,  a12t,...
      A20,  a21,      A22 ] = FLA_Repart_2x2_to_3x3( ATL, ATR,...
                                                     ABL, ABR, 1, 1, 'FLA_BR' );
    [ W00,  w01,      W02,...
      w10t, omega11,  w12t,...
      W20,  w21,      W22 ] = FLA_Repart_2x2_to_3x3( WTL, WTR,...
                                                     WBL, WBR, 1, 1, 'FLA_BR' );
    [ s0, ...
      beta1, ...
      s2 ] = FLA_Repart_2x1_to_3x1( sT, ...
                                    sB, 1, 'FLA_BOTTOM' );
%* ******************************************************************** *%
    u10t = FLA_Obj_set_R( a10t, 1 );
    alpha11 = alpha11 - u10t * w10t' - w10t * u10t';
    a21     = a21     - A20  * w10t' - W20 *  u10t';
    [ eta1, uhat, beta1 ] = HouseTrans( a21 );
    u = [ 1
          uhat ];
    a21 = [ eta1
            uhat ];
    z = beta1 * ( symml( A22 ) - A20 * W20' - W20 * A20') * u ;
    w21 = z - beta1/2 * (z' * u) * u;
%* ******************************************************************** *%
    [ ATL, ATR,...
      ABL, ABR ] = FLA_Cont_with_3x3_to_2x2( A00,  a01,      A02,...
                                             a10t, alpha11, a12t,...
                                             A20,  a21,      A22, 'FLA_TL' );
    [ WTL, WTR,...
      WBL, WBR ] = FLA_Cont_with_3x3_to_2x2( W00,  w01,      W02,...
                                             w10t, omega11,  w12t,...
                                             W20,  w21,      W22, 'FLA_TL' );
    [ sT, ...
      sB ] = FLA_Cont_with_3x1_to_2x1( s0, ...
                                       beta1, ...
                                       s2, 'FLA_TOP' );
  end
  A_out = [ ATL, ATR
            ABL, ABR ];
  W_out = [ WTL, WTR
            WBL, WBR ];
  s_out = [ sT
            sB ];
  return;
```

Figure 16: FLAME@lab code for accumulating an update for the blocked reduction to tridiagonal form algorithm.

```
function [ A_out, s_out ] = TriRedBlk( A, s, nb )

  [ ATL, ATR,...
    ABL, ABR     ] = FLA_Part_2x2( A, 0, 0, 'FLA_TL' );
  [ sT, ...
    sB ] = FLA_Part_2x1( s, 0, 'FLA_TOP' );

  while( size( ATL ) < size( A )-1 )
    b = min( nb, size( ABR, 1)-1 );

    [ s0, ...
      s1, ...
      s2 ] = FLA_Repart_2x1_to_3x1( sT, ...
                                    sB, b, 'FLA_BOTTOM' );

%* ********************************************************************** *%

  W = zeros( size( ABR, 1 ), b );

  [ ABR, W, s1 ] = AccumTriRed( ABR, W, s1 );

  [ A00, A01, A02,...
    A10, A11, A12,...
    A20, A21, A22 ] = FLA_Repart_2x2_to_3x3( ATL, ATR,...
                                             ABL, ABR,...
                                             b, b, 'FLA_BR' );
  [ W1, ...
    W2 ] = FLA_Part_2x1( W, b, 'FLA_TOP' );

  U21 = FLA_Obj_set_TR( A21, 1 );

  A22 = A22 - tril( U21 * W2' + W2 * U21' );

%* ********************************************************************** *%
    [ ATL, ATR,...
      ABL, ABR     ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02,...
                                                 A10, A11, A12,...
                                                 A20, A21, A22, 'FLA_TL' );
    [ sT, ...
      sB ] = FLA_Cont_with_3x1_to_2x1( s0, ...
                                       s1, ...
                                       s2, 'FLA_TOP' );
  end

  A_out = [ ATL, ATR
            ABL, ABR ];
  s_out = sT;

  return;
```

Figure 17: FLAME@lab code for blocked reduction to tridiagonal form algorithm.