LAPACK WORKING NOTE 195: SCALAPACK'S MRRR ALGORITHM

CHRISTOF VÖMEL*

Abstract. The sequential algorithm of Multiple Relatively Robust Representations, MRRR, can compute numerically orthogonal eigenvectors of an unreduced symmetric tridiagonal matrix $T \in \mathcal{R}^{n\times n}$ with $\mathcal{O}(n^2)$ cost.

This paper describes the design of ScaLAPACK's parallel MRRR algorithm. One emphasis is on the critical role of the representation tree in achieving both numerical accuracy and parallel scalability. A second point concerns the favorable properties of this code: subset computation, the use of static memory, and scalability.

Unlike ScaLAPACK's Divide & Conquer and QR, MRRR can compute subsets of eigenpairs at reduced cost. And in contrast to inverse iteration which can fail, it is guaranteed to produce a numerically satisfactory answer while maintaining memory scalability.

ParEig, the parallel MRRR algorithm for PLAPACK, uses dynamic memory allocation. This is avoided by our code at marginal additional cost. We also use a different representation tree criterion that allows for more accurate computation of the eigenvectors but can make parallelization more difficult.

AMS subject classifications. 65F15, 65Y15.

Key words. Symmetric eigenproblem, multiple relatively robust representations, ScaLAPACK, numerical software.

1. Introduction. Since 2005, the National Science Foundation has been funding an initiative [19] to improve the LAPACK [3] and ScaLAPACK [14] libraries for Numerical Linear Algebra computations. One of the ambitious goals of this initiative is to PUT MORE OF LAPACK INTO SCALAPACK, recognizing the gap between available sequential and parallel software support. In particular, parallelization of the MRRR algorithm [24, 50, 51, 26, 27, 28, 29], the topic of this paper, is identified as one key improvement to ScaLAPACK.

ScaLAPACK already provides several other symmetric eigensolvers, including QR algorithm, bisection and inverse iteration [23]), and the parallel Divide and Conquer method [60]. However, there are still good reasons for parallelizing MRRR.

1. MRRR allows the computation of subsets at reduced cost whereas QR and Divide & Conquer do not. For computing k eigenpairs of an $n \times n$ matrix on p processors, the tridiagonal parallel MRRR requires $\mathcal{O}(nk/p)$ operations per processor.
2. Inverse iteration does *not* guarantee a satisfactory answer with $\mathcal{O}(n^2/p)$ memory per processor. Depending on the spectrum of the matrix at hand, tridiagonal inverse iteration can require up to $\mathcal{O}(n^3)$ operations and $\mathcal{O}(n^2)$ memory *on a single processor* to guarantee a satisfactory set of computed eigenpairs. If the eigenvalues of the matrix are tightly clustered, the eigenvectors have to be computed (and re-orthogonalized against each other) on the same processor; parallelism in the inverse iteration is lost. MRRR is guaranteed to produce a satisfactory answer with $\mathcal{O}(n^2/p)$ memory and does not need re-orthogonalization.

*Christof Vömel, Institute of Computational Science, ETH Zürich, CAB, Universitätsstraße 6, CH-8092 Zürich, Switzerland, cvoemel@inf.ethz.ch.

This work has been partially supported by the National Science Foundation through the Cooperative Agreement no. ACI-9619020 and award no. CCF-0444486.

ParEig, another parallel version of the MRRR algorithm for the tridiagonal eigenvalue problem has already been developed [7]. The algorithm depends on PLAPACK [2, 62] for the orthogonal transformation to tridiagonal form and the backtransformation to obtain the eigenvectors of the original matrix. This excellent work invents the key concept of traversing MRRR's representation tree in parallel to guarantee orthogonality between eigenvectors on different processors. It is implemented in C and makes use of LAPACK 3.0 f77 kernels of MRRR. Memory is allocated dynamically as needed; MPI [35, 59] is used for parallel communication.

Our original motivation for parallelizing MRRR in the ScaLAPACK environment was to provide the same functionalities as ParEig, without requiring dynamic memory allocation. By its policy, ScaLAPACK only uses memory provided by the user. It is one contribution of this paper to show that indeed this is possible without substantial memory overhead. A second key difference to ParEig is the representation tree used for the computation. A comparison of a previous version of our code [4] in [67, 66] showed ParEig to be several times faster for a certain class of matrices. An investigation revealed the culprit [63]: one MRRR kernel from the new LAPACK 3.1 [44] was prone to generating long chains of large nodes in the representation tree, in contrast to the older LAPACK 3.0 kernel used by ParEig. The intent for changing this particular kernel in LAPACK 3.1 was to better prevent deteriorating orthogonality with increasing matrix size. Yet the implemented modification had the unintended bad side-effect of creating, for certain matrices of large dimension, artificially complex trees which where much more complicated to treat in parallel! A remedy to address this problem has been suggested in [63]. It is now implemented in our code and does cure the extreme behavior observed in [66].

This paper is organized as follows. The general ScaLAPACK approach to parallel eigencomputations is described in Section 2. Following the common three-phase approach, the parallel MRRR driver takes the dense distributed matrix at hand and transforms it to real symmetric tridiagonal form, solves the tridiagonal eigenvalue problem, and then applies the appropriate orthogonal/unitary transformations to obtain the eigenvectors of the original matrix.

The next three sections are the key part of this paper.

Section 3 gives a short overview of the MRRR algorithm, focusing on the role of the representation tree. It is exhibited how the requirement for numerical accuracy may stand in conflict with achieving parallel scalability. Various parallelization strategies with their benefits and shortcomings are discussed. The importance of a homogeneous computing environment for parallel correctness is highlighted.

Section 4 consists of a comparison of the differences between ScaLAPACK's MRRR and ParEig and further illuminates the impact of the different representation trees on accuracy, performance, and scalability.

Section 5 presents a performance and scalability analysis of the tridiagonal part of ScaLAPACK's MRRR. As comparison, results with parallel Divide & Conquer are shown.

Then, conclusions and ideas for future work are presented. Appendix A describes the design of the ScaLAPACK tester. Appendix B contains examples that illustrate the differences between the parallelization strategies in Section 3.2. Appendix C compares the interfaces of parallel MRRR and bisection/inverse iteration. Appendix D considers applications of this current work, focusing on electronic structure calculations.

2. Outline of the parallel design. The MRRR ScaLAPACK distribution consists of four drivers. To keep the description simple, the presentation in this paper focuses on the real double precision driver PDSYEVR. There also are drivers called PSSYEVR, PCHEEVR, and PZHEEVR for real single and complex Hermitian single and double precision, respectively. From the user's point of view, the only difference between the real and complex versions is that storing complex numbers needs twice the amount of memory required for the corresponding real number. The code does exploit that every Hermitian matrix can be reduced to real tridiagonal form. Thus, the internal MRRR kernels are required in real format only.

2.1. The existing ScaLAPACK environment. This section gives a short overview of the ScaLAPACK environment to explain its philosophy and constraints. For a more complete description, see [54, 14].

2.1.1. ScaLAPACK, BLACS, and PBLAS. Except for the (significant) additional complexity of parallel communication, ScaLAPACK [14] algorithms look similar to their LAPACK [3] counterparts. Both LAPACK and ScaLAPACK rely heavily on block-partitioned algorithms. In principle, ScaLAPACK uses two fundamental building blocks.

- The Parallel Basic Linear Algebra Subprograms, PBLAS [16] are distributed-memory versions of the BLAS [9, 31, 32, 45].
- The Basic Linear Algebra Communication Subprograms, BLACS [30, 33] provide interfaces for common communication tasks that arise frequently in parallel linear algebra computations.

Furthermore, on individual processors, ScaLAPACK makes frequent use of the available LAPACK computational kernels or slight modifications of them.

As an example, Algorithm 1 describes the principal structure of PDSYEVX, the ScaLAPACK expert driver for the symmetric eigenvalue problem. It has the same structure as the respective sequential LAPACK expert driver DSYEVX, solely the prefix 'P' indicates that the called subroutines work on distributed data. As usual, the eigenvalue problem is solved not using A but an equivalent tridiagonal T. * First, the eigenvalues are computed by bisection [18]. Afterwards, the corresponding eigenvectors are computed by inverse iteration [25, 38, 39].

Algorithm 1 Outline of ScaLAPACK's driver PDSYEVX based on parallel bisection and inverse iteration.

Input: distributed symmetric matrix A and a range of values or indices of desired eigenvalues
Output: eigenvalues W and distributed eigenvector matrix Z
(1) Transform A into tridiagonal form $Q \cdot T \cdot Q^T$ (PDSYNTRD).
(2) Broadcast tridiagonal matrix T to all processors.
(3) Compute eigenvalues and eigenvectors of T in parallel (PDSTEBZ and PDSTEIN).
(4) Apply distributed orthogonal matrix Q to eigenvectors of T to obtain the distributed Z containing the eigenvectors of A (PDORMTR).

2.1.2. Memory management. Currently, both LAPACK and ScaLAPACK only use memory provided by the user; no memory is allocated internally. The required workspace is an explicit part of the interface, a program will not run unless enough

*Note the use of PDSYNTRD in Algorithm 1. It is an update to PDSYTRD [15] to improve scalability.

memory is supplied. For convenience, a user can issue a query to find out how much work space is needed for a given problem (and processor configuration) and then allocate the required amount.

There are two major benefits of this policy. First, the user has complete control over memory management. No LAPACK or ScaLAPACK library function aborts unexpectedly due to insufficient memory when dynamic allocation fails. Second, there is no unknown run-time cost in terms of memory and time on each allocation and deallocation in the libraries.

The algorithm developer however has to carefully manage and reuse the provided workspace which can be tedious prone to errors that are hard to debug. Another concern is to anticipate potential future algorithmic changes that might require a larger amount of workspace from the user and thus changes in users' codes.

2.1.3. Communication management. The BLACS provide only synchronous send and receive routines for communication [14]. Thus, each communication implicitly serves as a barrier. Furthermore, this implies that calls to sequential LAPACK codes cannot be overlapped with BLACS-based communication. Depending on the application, this can be less efficient than the use of non-blocking communication.

2.2. The new driver pdsyevr. The design of PDSYEVR is given as Algorithm 2. It is very similar to the existing ScaLAPACK driver PDSYEVX shown in Algorithm 1. The difference lies in the way communication is handled. Whereas part of the communication in PDSYEVX is hidden inside of PDSTEBZ and PDSTEIN, we decided to gather all communication in PDSYEVR itself.

Algorithm 2 Outline of the new ScaLAPACK driver PDSYEVR based on the MRRR algorithm.

Input: distributed symmetric matrix A and a range of values or indices of desired eigenvalues
Output: eigenvalues W and distributed eigenvector matrix Z
(1) Transform A into tridiagonal form $Q \cdot T \cdot Q^T$ (PDSYNTRD).
(2) Broadcast tridiagonal matrix T to all processors.
(3) Compute eigenvalues and eigenvectors in parallel using an MRRR-based kernel.
(4a) Broadcast the computed eigenvalues to all processors.
(4b) Distribute the eigenvectors into Z (PDLAEVSWP).
(5) Apply distributed orthogonal matrix Q to eigenvectors to obtain the distributed Z containing the eigenvectors of A (PDORMTR).

The key part is (3): based on some division of work, the processors compute the wanted eigenvalues and subsequently their eigenvectors in parallel; details are given in Section 3. Then the eigenvalues are shared so that they are available on all processors(4a), and the eigenvector matrix is distributed (4b). As the input matrix A is stored in ScaLAPACK's 2D-block-cyclic layout, the eigenvector matrix Z uses 2D-block-cyclic layout as well. This is required by ScaLAPACK to make the orthogonal matrix-matrix multiply in step (5) efficient. However, MRRR computes the eigenvector matrix column-wise, that is, all of an eigenvector is computed on a single processor. For this reason, the eigenvectors are distributed in 1D form across the processors, and step (4b) is responsible for the redistribution of the eigenvector data from 1D into 2D form.

3. A road map for the computational MRRR-based kernel. This section describes the heart of the new ScaLAPACK driver PDSYEVR, that is, step (3) from Algorithm 2 in Section 2.2. To set the stage, Section 3.1 gives a short overview of the MRRR algorithm. The emphasis is on introducing the representation tree as principal concept governing the eigenvector computation. Afterwards, in Section 3.2, we discuss different ways of using it in a parallel environment.

3.1. The MRRR algorithm and the representation tree. This section presents a very short introduction to the MRRR algorithm in order to show different sources of parallelism in the computation. More details can be found in [29, 50, 51, 26, 27].

For simplicity of presentation, we assume the tridiagonal $T \in \mathcal{R}^{n\times n}$ to be unreduced: all its off-diagonal entries are nonzero [48, Definition 7.1.2]. This is not an algorithmic restriction, we merely prefer not to introduce additional notation for unreduced sub-blocks. In practice, MRRR scans the matrix for unreduced blocks and works separately on each of them: this decreases the complexity of the computation and creates additional parallelism. Note also that the eigenvalues of an unreduced matrix, in exact arithmetic, are simple [48, Lemma 7.7.1]. Nevertheless, the task of computing orthogonal eigenvectors without resorting to Gram-Schmidt is still a difficult one: in [28] it is shown that despite being unreduced and without any off-diagonal entry being particularly small, a tridiagonal can have eigenvalues that are indistinguishable when being represented as floating point numbers.

In order to compute orthogonal eigenvectors, one of MRRR's goals is to find an approximation $\hat{\lambda}$ to each desired true eigenvalue λ so that

$$|\lambda - \hat{\lambda}| = \mathcal{O}(\epsilon|\lambda|). \tag{3.1}$$

Here, ϵ refers to the relative machine precision. The interpretation of (3.1) is that $\hat{\lambda}$ has high *relative* accuracy.

However, in finite precision, the original tridiagonal T need not define its eigenvalues to high relative accuracy: small relative changes in the entries of T can cause much larger changes in its eigenvalues, see [48, Section 2.7] and [49]. It is known from backward error analysis [18] that the finite-precision computation of $\hat{\lambda}$ from T is associated with such small componentwise changes. Hence, requiring (3.1) to hold for the eigenvalues of T may be unrealistic. For this reason (and others that will become clear shortly), the algorithm replaces the original tridiagonal matrix T by a factorization

$$LDL^T = T - \sigma I. \tag{3.2}$$

Here $L \in \mathcal{R}^{n\times n}$ is a lower bidiagonal matrix with unit diagonal, and $D \in \mathcal{R}^{n\times n}$ is diagonal containing the pivots from Gaussian elimination. One requires from the shift σ to yield an LDL^T where small relative changes in entries of L and D cause small relative changes in some or all of its eigenvalues. Such an LDL^T factorization is called a Relatively Robust Representation (RRR) for those of its eigenvalues that are defined to high relative accuracy. When σ is chosen such that LDL^T becomes definite, it can be computed stably without element growth and it is an RRR for all its eigenvalues, see [20, 52].

As LDL^T is thus better suited for finite precision computation, MRRR does not work with T except to compute (3.2). For each desired eigenvalue λ of LDL^T with an eigenvalue approximation $\hat{\lambda}$ satisfying (3.1), one can hope to also compute an

approximate eigenvector v with a residual norm small compared with $|\lambda|$ instead of $\|LDL^T\|_2$:

$$\|(LDL^T - \hat{\lambda}I)v\| = \mathcal{O}(n\epsilon|\lambda|). \tag{3.3}$$

Note that the small relative residual (3.3) can only be achieved with respect to LDL^T and not with respect to T. Note that the left-hand side $|\lambda - \hat{\lambda}|$ of (3.1) constitutes a lower bound on the residual norm (3.3) of the computed vector v. Thus, the right-hand side of (3.1) can only hold with LDL^T, as T does not allow an eigenvalue approximation to high relative accuracy.

At this point, the classical gap theorem [17, 48] implies

$$|\sin\angle(v,z)| \leq \frac{\|(LDL^T - \hat{\lambda}I)v\|}{\mathrm{gap}(\hat{\lambda})} =: \frac{\mathcal{O}(n\epsilon)}{\mathrm{relgap}(\hat{\lambda})}. \tag{3.4}$$

where z denotes the true eigenvector of LDL^T from (3.2). The gap of $\hat{\lambda}$ approximating the true λ is defined as

$$\mathrm{gap}(\hat{\lambda}) = \min\left\{|\hat{\lambda} - \mu| : \lambda \neq \mu, \mu \in \mathrm{spectrum}(LDL^T)\right\}, \tag{3.5}$$

and the relative gap as $\mathrm{relgap}(\hat{\lambda}) = {}^{\mathrm{gap}(\hat{\lambda})}/_{|\hat{\lambda}|}$. It is noteworthy that in (3.4), it is the *relative* gap, not the usual standard gap as in [48, Theorem 11.7.1], that governs the deviation from orthogonality. One defines a *singleton* to be a shifted eigenvalue approximation satisfying (3.1) and

$$\mathrm{gap}(\hat{\lambda}) \geq \tau_{\mathrm{minrgp}} \cdot |\hat{\lambda}|, \tag{3.6}$$

that is, $\mathrm{relgap}(\hat{\lambda}) \geq \tau_{\mathrm{minrgp}}$ when $\hat{\lambda} \neq 0$. Here, τ_{minrgp} is a threshold that specifies the smallest admissible relative gap. Admissibility is to be understood in terms of allowable orthogonality loss: by (3.4), MRRR can guarantee that the corresponding eigenvector satisfies

$$|\sin\angle(v,z)| = \frac{\mathcal{O}(n\epsilon)}{\tau_{\mathrm{minrgp}}}. \tag{3.7}$$

The smaller the threshold, the higher is the potential deviation from orthogonality between different computed vectors each satisfying (3.7). Section 4 discusses different choices for τ_{minrgp} and their impact on accuracy, performance, and parallelism.

Once an approximation $\hat{\lambda}$ to high relative accuracy for each λ has been computed, and provided that the relative gaps are large enough, MRRR can also compute each approximate eigenvector v with small relative residual and a small angle to its corresponding true eigenvector. This guarantees orthogonality between the computed vectors without Gram-Schmidt. What then remains is to ensure an acceptable residual with respect to T. It is shown in [26] that because of the stable computation of (3.2) without element growth, the approximate eigenpair $(\hat{\lambda} + \sigma, v)$ will satisfy $\|(T - (\hat{\lambda} + \sigma)I)v\| = \mathcal{O}(n\epsilon\|T\|)$.

We now turn to the realistic complication of LDL^T having clustered eigenvalues such that not all relative gaps are large enough to pass (3.6). For this case, MRRR exploits that the relative gaps depend on the magnitude of the eigenvalue approximation. For a cluster with eigenvalues that are each not passing the test (3.6), a new

RRR is computed using a shift that is close to the cluster. The differential stationary qd algorithm *dstqds* [47] provides a stable way of computing the new RRR

$$L_+D_+L_+^T = LDL^T - \sigma' I. \tag{3.8}$$

'Stable' here means a mixed relative stable factorization: small relative componentwise perturbations in the entries of D, L and D_+, L_+ give an exact shift relation, see [27, Theorem 2 and Figure 3]. The choice of the incremental shift σ' for the new RRR is of key importance. One needs to guarantee that [26]

- the new factorization $L_+D_+L_+^T$ is an RRR for all eigenvalues of the group, and that
- at least one eigenvalue of the group, the one closest to the shift, becomes a singleton.

The former requirement on the RRR property guarantees that the local eigenvalues of $L_+D_+L_+^T$ remain meaningful. Note that by shifting close to the cluster, the eigenvalues of $L_+D_+L_+^T$ are smaller in magnitude than the corresponding ones from LDL^T and thus the relative gaps become larger. For the latter requirement, finding at least one singleton per RRR is not only motivated by efficiency but also by correctness: it guarantees a finite procedure with a limited number of RRRs and limited error growth, see [26]. (In practice, the algorithm usually finds more than one singleton per RRR which is key to its efficiency.) In order to guarantee that at least one singleton will be found, one has to shift close enough so that subsequent eigenvalue refinement for the new RRR can reveal a large enough relative gap. Furthermore, one prefers to shift just outside an eigenvalue group rather than inside as to reduce the risk of element growth in the factorization which could spoil the RRR property.

Element growth in a factorization can be dangerous as componentwise small relative perturbation in the entries of D and L will result large absolute perturbations to $L_+D_+L_+^T$. Thus, MRRR samples different locations for finding a factorization with small element growth. It evaluates shifts at both ends of the cluster and also backs off the cluster ends if necessary, see [29]. (It is interesting to note that even if element growth occurs, it need not destroy the RRR property of the factorization for the eigenvalues in the cluster, see [24, Example 5.1.2].)

The MRRR procedure can be described via the *representation tree* [24, 26, 63]. Each node of the tree represents the RRR for a group of eigenvalues that is separated from its neighbors but whose eigenvalues are close in the sense of relative gaps. The root node is the initial representation and an RRR for all the wanted eigenvalues, it is called the root representation. Each child of a parent is either a singleton (and thus a leaf of the tree), or an RRR for a subset of clustered eigenvalues of the parent. The edge between a parent and a (non-leaf) child stands for a *dstqds* computation (3.8). A simplified summary of the MRRR procedure is given in Algorithm 3. An example of a representation tree will be shown in Figure 3.1 of Section 3.2.1.

An investigation of the different steps of Algorithm 3 yields the following sources of potential parallelism.

1. *Root representation:* the computation of the factorization costs $\mathcal{O}(n)$ operations and is essentially sequential. However, the computation of all wanted eigenvalues by bisection is an embarrassingly parallel operation: each individual eigenvalue can be computed independently from all the others.
2. *General RRR:* the computation of a child RRR from its parent is again essentially sequential. However, in order to detect large relative gaps in an RRR, eigenvalues need to be refined by bisection. This is again embarrassingly parallel.

Algorithm 3 Outline of the (sequential) MRRR algorithm.

(1) For (unreduced) T, compute a root representation $LDL^T = T - \sigma I$ and its wanted eigenvalues (user-specified by range of values or indices)
(2) Build the representation tree. Using a series of incremental shifts and qd transformations of the form (3.8), find suitable RRRs so that, eventually, all eigenvalues become singletons.
(3) For each singleton, compute the corresponding eigenvector so that (3.3) holds. This guarantees orthogonality by (3.4).

3. *Singleton:* as soon as an RRR is known for which a local eigenvalue is a singleton, the eigenvector computation becomes independent from the computation of all other eigenvectors. Thus, once such an RRR is known for each eigenvector, the computation of all eigenvectors is embarrassingly parallel.

3.2. Setting up the MRRR-based parallel kernel.

3.2.1. Subset-based parallelization. Given a set of $k \leq n$ wanted eigenpairs, a straightforward idea for parallelization is to assign non-overlapping subsets of k/p eigenpairs to each of the p processors. Then, the subset feature [46] of the sequential code (Algorithm 3) could be invoked to compute on each processor the assigned k/p eigenpairs. As no communication or synchronization between the processors is involved, this approach is perfect with respect to complexity and memory.

Even though this approach has been used in ScaLAPACK's driver PDSYEVX, it is too simplistic and does not guarantee numerical accuracy without resorting to Gram-Schmidt which spoils memory scalability, see the later Section 3.3.

If the sequential MRRR code is called just on the subset of k/p eigenpairs assigned to the processor, it returns eigenpairs that have small residuals and eigenvectors that are mutually orthogonal *among each other*. No orthogonality is guaranteed between computed eigenvectors from different non-overlapping subsets. [†]

The orthogonality problem is reflected in different representation trees. Figure 3.1 shows the representation tree of a sample matrix when all eigenpairs are computed on a single processor.

Now assume that the computation were parallelized among four processors using the subset approach. As each processor only regards its part of the assigned spectrum, it derives a representation tree for only its own eigenpairs, these are shown in Figure 3.2. However, by ignoring the rest of the spectrum, only the eigenvectors within each processor are guaranteed to be orthogonal. Orthogonality across processors depends on the cluster structure and is *not* guaranteed as (3.6) may be violated between the sets assigned to different processors. An example for a failure of subset-based parallelization is given in Appendix B.1.

Note that it would be possible to allow overlapping subsets and make each processor perform redundant computation in order to guarantee the use of 'consistent'

[†]ScaLAPACK's driver PDSYEVX calls bisection and inverse iteration on non-overlapping subsets for parallelization. There is no guarantee for orthogonal eigenvectors unless reorthogonalization is performed inside tight clusters. For reorthogonalization to be done for a given cluster, all eigenvectors must be computed on the *same* processor. Thus, PDSYEVX cannot guarantee the right answer unless the user supplies enough memory. In the most extreme case, all or almost all wanted eigenvectors have to be computed on a single processor; memory scalability has to be sacrificed to guarantee orthogonal vectors. Note that the current ScaLAPACK tester does not report poor orthogonality as a failure when it stems from insufficient memory, see Appendix A.3.

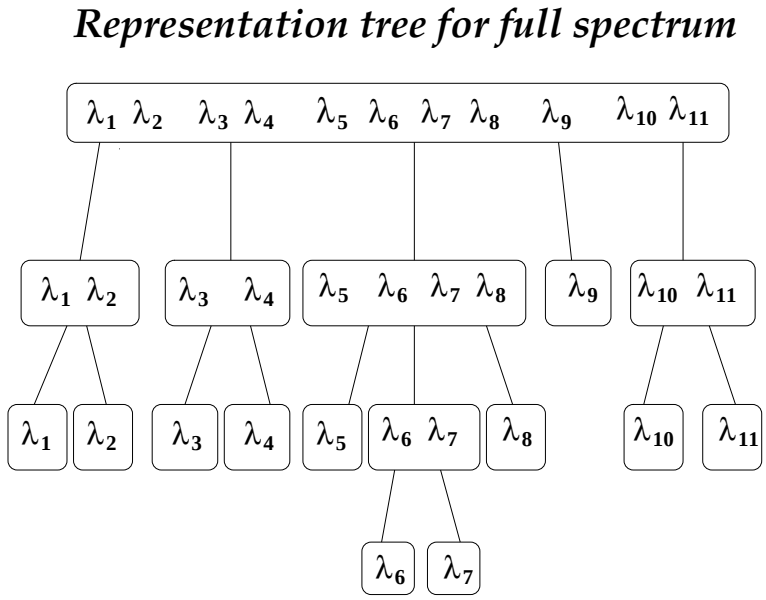

FIG. 3.1. *Illustration of a (sequential) representation tree for computing all eigenpairs of a matrix of dimension* 11. *Square boxes correspond to singletons, rectangular ones correspond to eigenvalue groups for which an individual RRR is needed to improve relative gaps. Upon inspection of the root RRR (top level) for large relative gaps, MRRR finds that a new RRR needs to be computed for* λ_1, λ_2, *for* λ_3, λ_4, *for* $\lambda_5, \ldots \lambda_8$, *and for* $\lambda_{10}, \lambda_{11}$. λ_9 *is a singleton with respect to the root RRR. At the next deeper level, inspection of the RRRs reveals that all local eigenvalues are now singletons, except for* λ_6, λ_7. *For these, one more RRR needs to be computed.*

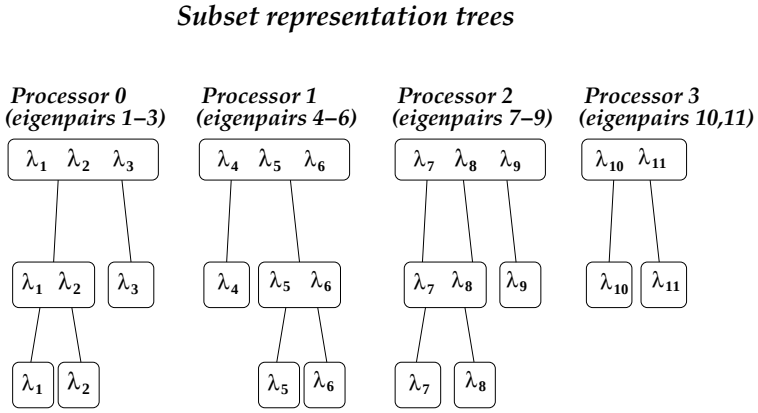

FIG. 3.2. *Illustration: each processor computes a subset of the wanted eigenpairs for the matrix from Figure 3.1. This approach does not guarantee orthogonality of eigenvectors computed on different processors.*

root representations on all processors. This approach will work, according to the classical gap theorem (3.4), provided an *isolated* superset of eigenvalues is known for each processor. However, there need not be a *small* isolated superset; in the worst case, each processor would have to compute all of the eigenvalues to some accuracy. Thus, in none of its forms, does the subset based approach guarantee both accuracy and scalability at the same time.

3.2.2. Parallelization of the eigenvector computation via conformal embedding of subset representation trees. In this section, we suggest an approach that achieves the desired $\mathcal{O}(nk/p)$ storage per processor (for k wanted eigenpairs) while at the same time guaranteeing orthogonal eigenvectors. This strategy implements the idea proposed in [7] for a parallel traversal of the representation tree.

The algorithm starts with a superset of the wanted eigenvalues that is isolated from the rest of the spectrum. For simplicity of presentation, we assume it to be the full spectrum. Then, the *relevant* part of the full representation tree is constructed. All those representations are computed that define *at least* one of the wanted eigenvalues. The procedure is then repeated. The computation of irrelevant representations as well as of unwanted eigenvectors is omitted.

When used in parallel (as step (2) of Algorithm 3), this procedure yields *consistent* representation trees among all processors, that is, all processors start from the same root. Consequentially the parallel algorithm does produce mutually orthogonal sets of eigenvectors on different processors. In Figure 3.3, we illustrate this approach by the example from Figure 3.1 and the eigenpairs $4-6$ assigned to processor 1 (processor numbering and subset assignment according to Figure 3.2).

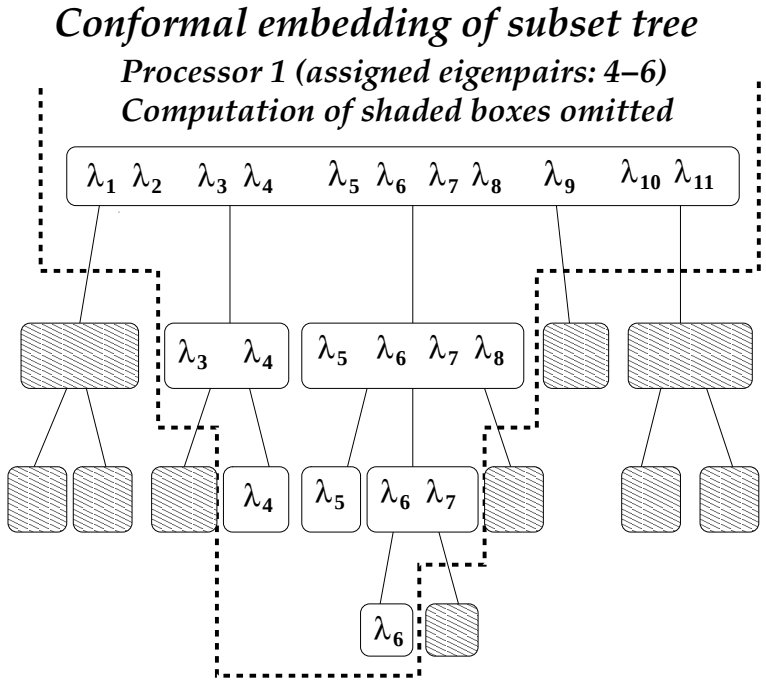

FIG. 3.3. *Illustration of the conformal embedding of the subset representation tree into the representation tree for the full spectrum. (Compare to Figures 3.1 and 3.2.)*

3.2.3. Implementation with static memory. For the ScaLAPACK implementation with fixed memory, the question of how to store intermediate representations needs to be addressed. In the sequential case, RRRs for clusters are stored in the eigenvector matrix Z and later overwritten by the eigenvectors. This is based on the observation that one only needs to compute a new RRR when there are at least two clustered eigenvalues. Thus, there always is enough space in the eigenvector matrix to store the entries of the RRR, see also [29].

We recently found that it is possible to store the RRRs in the parallel case with only one additional vector on each side of the wanted spectrum. The additional difficulty concerns the case when an extremal wanted eigenvalue is part of a cluster whose other eigenvalues do not belong to the wanted spectrum. In this case, we need to compute an RRR for the single eigenvalue. This requires $2n$ storage of which only n are available from the usual eigenvector storage. This is illustrated in Figure 3.4.

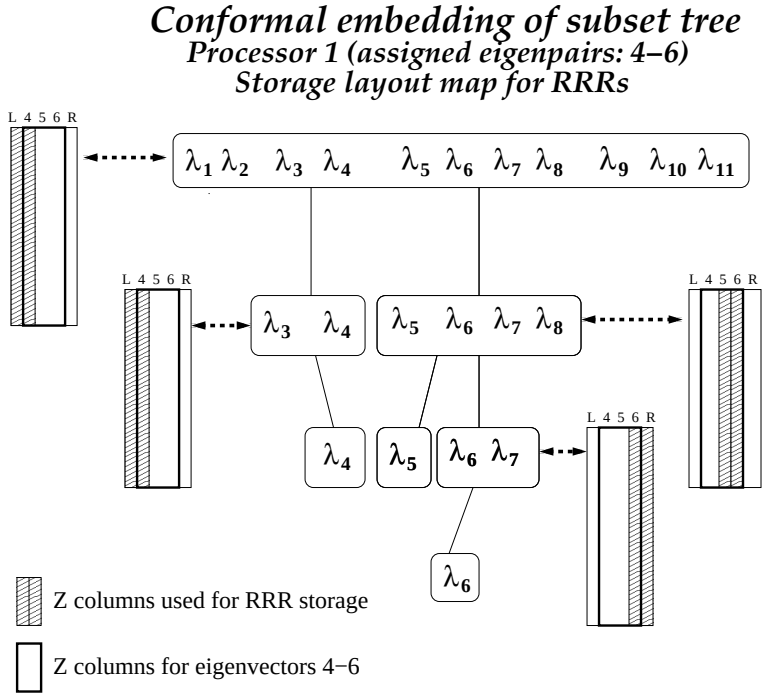

FIG. 3.4. *Storage map for the RRRs in the eigenvector matrix when the subset tree is embedded.*

Note that the use of fixed storage does not incur a performance penalty. The RRRs are stored in a continuous chunk of memory that corresponds to the way the RRRs are stored in the sequential code. Further, there is no overhead in data structure reorganization: as in the sequential case, the RRR is stored in a fixed place in the array of the eigenvectors and a vector copy operation is used for retrieving it.

One issue remains. The algorithm requires a superset of the wanted eigenvalues that is well isolated. Depending on the matrix, this superset can be substantially larger than the k/p eigenvalues one would ideally like to work with; in the extreme case it can contain all k wanted eigenvalues on each processor. This does not jeopardize the important memory scalability, each processor still requires $\mathcal{O}(nk/p)$ memory. However, the worst case bound for the computational complexity is $\mathcal{O}(nk)$ instead of the optimal $\mathcal{O}(nk/p)$ unless the eigenvalue computation (step (1) of Algorithm 3) is parallelized, too. (In the context of the dense symmetric eigenvalue problem, the $\mathcal{O}(nk)$ complexity of the tridiagonal eigensolver is a cosmetic imperfection because the transformation to tridiagonal form is $\mathcal{O}(n^3/p)$ at best.)

3.2.4. Parallelization of the eigenvalue computation. The sequential MRRR algorithm uses dqds [52] to compute the eigenvalues of the root representation. This is an $\mathcal{O}(n^2)$ process. Consequently, bisection is preferable in the case of a large matrix on

many processors to prevent the eigenvalue computation from becoming a bottleneck.

In this section, we describe how to achieve $\mathcal{O}(nk/p)$ complexity in step (1) of Algorithm 3, when computing k eigenpairs. The following simple procedure (almost) solves the problem:

- Call bisection in parallel on the wanted part of the spectrum.
- Send the eigenvalues to all processors that need them.

However, the procedure does not take into account that the code always computes eigenvalues of a shifted root representation LDL^T and not the original matrix. Thus, it has to be ensured that different processors use a consistent root representation when eigenvalues are computed in parallel. This yields the following modified procedure for each processor:

- The RRR for the cluster at hand is computed redundantly.
- Bisection is called to refine a subset of eigenvalues from the cluster.
- The refined eigenvalues and their uncertainties are broadcast to all processors that need them (synchronization between the subset of processors responsible for the cluster at hand).
- Other information such as relative gaps is redundantly recomputed from the refined eigenvalues. The computation of the representation tree and the eigenvectors now proceeds independently from the other processors.

The same idea of parallelizing the eigenvalue computation between processors can be used not only for the root node but also for RRRs further down in the representation tree. Using this feature results in finer-grained parallelism for the representation tree. It reduces the amount of redundant computations at the price of increased communication.

3.2.5. Requirements of the computing environment. In [8], the authors identify the following key aspects of homogeneity:

- Hardware (processors) and software (compiler and operating system) store floating point numbers in the same way and produce the same results for floating point operations.
- Communication transmits floating point numbers exactly.

Heterogeneous computing poses enormous challenges for some of the ScaLAPACK algorithms. In [18], the authors devise ways to ensure the correctness of parallel bisection in a non-homogeneous environment.

At this point, we only have practical experience with MRRR in a homogeneous environment. Since MRRR is much more complicated than bisection, it is highly non-trivial to find criteria for its parallel correctness. For this reason, we advise the use of the parallel MRRR algorithm only in an environment that guarantees the computation to be consistent with the sequential code.

3.3. Usage of memory. A common question a user faces is the question of the maximum matrix size that an algorithm permits with the amount of memory available on a system. The answer depends on the user data as well as the system architecture and the compiler. For this reason, it is most convenient to describe memory usage in terms of data volume. To find the required memory in bytes, one would multiply the data amount by the storage costs per data. [‡]

For the computation of all eigenpairs the memory requirements for PDSYEVR are $\lceil 4n^2/p \rceil$ per processor, this includes space for the matrices A and Z. When A is

[‡]For example, a system might store an integer as 4 bytes, a real single and double precision number in 4 and 8 bytes respectively, with the respective complex numbers being twice as costly.

complex Hermitian, half of this amount is needed as real, the other half as complex data. For the computation of a subset of k eigenpairs, the code needs $\lceil (n^2 + 3kn)/p \rceil$ memory per processor. For complex A, $\lceil 2kn/p \rceil$ of this are real numbers. [§]

Algorithm 2 requires storage for the input matrix A and its eigenvector matrix Z. Both are of the same type, real or complex. The orthogonal reduction of A to tridiagonal form T is stored by overwriting the input matrix A. However, ScaLAPACK does not currently support packed storage for symmetric matrices, and thus A is also associated to n^2 storage. Thus the lower bound on storage per processor for the full eigenproblem in ScaLAPACK is given by $\lceil 2n^2/p \rceil$.

The eigenvector matrix of the tridiagonal is always real. However, for higher efficiency of the matrix-matrix multiplication in the backtransformation of the eigenvectors of T to the eigenvectors of A, there is one additional memory-intensive step in Algorithm 2. Z is stored using ScaLAPACK's 2D-block-cyclic layout but the eigenvector computation requires 1-D layout (see Section 2.2). Thus, the code requires additional $\lceil n^2/p \rceil$ intermediate workspace per processor for storing the eigenvectors column-wise. The remaining $\lceil n^2/p \rceil$ workspace are used as communication buffer by the redistribution routine PDLAEVSWP mentioned in Section 2.2: for each pair of processors, it sends all the information from the 1D grid in one shot, thus the large worst-case memory bound.

Compared to bisection and inverse iteration PDSYEVX, one can see that the work space roughly equals the minimum required work space. On the other hand, when Gram-Schmidt orthogonalization is required, the *additional* memory required by PDSYEVX on a single processor can become almost as large as the eigenvector matrix itself. Thus, MRRR's avoidance of Gram-Schmidt additionally pays off by memory scalability in the parallel case.

4. Performance role of the representation tree and comparison with ParEig. This section illuminates the difference between the representation trees used by ParEig and PDSYEVR. This comparison sheds light on some results published in [66].

4.1. Runtime scalability: easy and hard cases. To set the stage, we consider two model problems that illustrate how the performance and scalability of MRRR can vary depending on the representation tree at hand.

1. *(The easy case for MRRR.)* Consider a tridiagonal matrix with a shallow representation tree, consisting of only one root node whose children are all relatively isolated, that is singletons. In this ideal case, each processor can directly compute its part of the eigenvectors from the root RRR, the computation is embarrassingly parallel.
2. *(The hard case for MRRR.)* Consider a tridiagonal matrix whose eigenvalues are all strongly clustered. If the clustering can only be resolved gradually, that is clusters within clusters are present, the representation tree becomes deep. Even though the eigenvector computation itself is fully parallelized, the construction of the representation tree requires a substantial overhead that is redundant on each processor before the eigenvectors can be computed. This overhead mainly consists of repeated eigenvalue refinement that may also involve communication between processors.

[§] Additional real and integer workspaces of respective sizes about $25n$ and $14n$ per processor are required for the MRRR-based computational kernel and storage of intermediate data. This is independent of the number of eigenpairs to be computed.

In general, a representation tree at hand may lie somewhere in between these two extreme cases. The part of the spectrum that is well spread out like in the first case can be dealt with easily. If there is a part of the spectrum that is clustered as in the second case, the workload associated with this part of the representation tree is more significant.

While the processors do the same amount of computation for the eigenvectors of the singletons, the amount of work for constructing the intermediate parts of the representation tree can be substantially different. A processor with a deep part of the tree needs to compute more intermediate representations and their eigenvalues. Consequently, load imbalance can occur and hamper the overall performance of the code.

4.2. Clustering in MRRR. The issue that is of particular importance for the following is how MRRR groups those eigenvalues that are *not* singletons. The approach that is used in LAPACK 3.0 [3] and 3.1 [44] is to inspect one-sided gaps. This was motivated as follows. If an eigenvalue has large relative gap on its left-hand side but not its right, it seemed natural to consider the eigenvalue the left end of a new cluster. Vice-versa, with a small left but a large right relative gap, an eigenvalue can be considered the right end of a cluster. In this case, (3.6) which requires for a singleton to have large enough relative gaps on both sides, characterizes singletons as clusters of size one.

Consider a cluster with given RRR and eigenvalue approximations. Starting from the left of a cluster, the right gap $\mathrm{rgap}(\hat{\lambda})$ of each eigenvalue is inspected. Then a boundary from one child to the next is defined through the separation criterion

$$\mathrm{rgap}(\hat{\lambda}) \geq \tau_{\mathrm{minrgp}} \cdot |\hat{\lambda}|, \tag{4.1}$$

where $\hat{\lambda}$ approximates a local eigenvalue of the RRR.

If multiple eigenvalues are enclosed between rgaps satisfying (4.1), one has found a group for which a new RRR is computed. Otherwise one has found a singleton.

The following section describes two choices, the ones used in LAPACK 3.0 and 3.1, for the minimum relative gap threshold in (4.1). The point of the discussion is to show that the thresholds can produce very different representation trees. ParEig [7] uses the LAPACK 3.0 criterion. The previous version [4] of PDSYEVR that was evaluated in [66] used the LAPACK 3.1 criterion.

4.3. Two choices for $\tau_{\mathbf{minrgp}}$. Based on a careful analysis of the propagation of rounding errors in the representation tree [24, 27], the initial LAPACK 3.0 implementation of the MRRR algorithm used

$$\tau_{\mathrm{minrgp}}^{\text{LAPACK 3.0}} := \min\left\{10^{-2}, 1/n\right\}, \tag{4.2}$$

with n being the dimension of the (unreduced) tridiagonal matrix.

While the (pessimistic) bound from (3.4) would predict a loss of orthogonality proportional to n^2, a more careful analysis shows that using (4.2), the loss of orthogonality should be proportional to $n\sqrt{n}$ at worst. Moreover, based on experimental evidence, until around 2004 it was believed that residual norms and dot products between different computed vectors were bounded by multiples of $n\epsilon$.

While working on the latest version of MRRR [29] for the new LAPACK 3.1 and more extensive testing on much larger matrices with more and more challenging eigenvalue distributions, it was felt that the threshold should be a constant to better

prevent deteriorating orthogonality with larger matrices that was observed in some experiments. Another motivation for changing the threshold was to make the accuracy threshold independent from the matrix size. For this reason, the LAPACK 3.1 release [44] uses a threshold independent of n,

$$\tau_{\text{minrgp}}^{\text{LAPACK 3.1}} := 10^{-3} \tag{4.3}$$

that is, eigenvalues agreeing to less than three digits are declared relatively isolated. As fewer relative gaps pass threshold test (4.3), the clustering, with respect to the new criterion, is stronger and the representation tree deeper.

For matrices of dimension larger than one thousand, a comparison [22] of LAPACK's tridiagonal eigensolvers using a comprehensive set of test problems [21] shows improvements in the accuracy of MRRR in LAPACK 3.1 compared to version 3.0. These were attributed in part to the modified threshold τ_{minrgp}. In [66], one can also see the worse accuracy of ParEig compared to PDSYEVR.

4.4. The peeling problem and how it can be addressed. [66] showed that PDSYEVR, while computing eigenvectors with better orthogonality, was several times slower than ParEig on matrices from Hubbard models in electronic structure calculations. We analyze the reason by the example of the largest Hubbard matrix, n=63504.

Let the eigenvalues be numbered in ascending order from the left of the spectrum to the right, then an analysis of the representation tree for PDSYEVR, using LAPACK 3.1 criterion (4.3), reveals the existence of a cluster of size 61735 (!), spanning eigenvalues 1115 to 62849, that contains a cluster of eigenvalues 1115 to 59566, in which eigenvalues 4226 to 59566 form again a cluster, and so forth. The construction of the representation tree thus involves a severe overhead in repeated eigenvalue refinement. On the other hand, using the old LAPACK 3.0 criterion (4.2), there are no clusters of size larger than four! Obviously, the corresponding representation tree offers very fine grain natural parallelism. This is reflected in big runtime differences: on 32 processors on the IBM SP5, the tridiagonal part of PDSYEVR using criterion (4.2) takes 237 seconds, versus 884 seconds using criterion (4.3). Moreover, scalability for the deep tree is terrible but great for the shallow one, see also [66].

From the parallelism and efficiency point of view, the tree used by ParEig is clearly preferable. Does the relatively small extra amount of accuracy obtained by the PDSYEVR tree justify all the additional work? Further inspection of the eigenvalue spectrum revealed that the answer is no. The eigenvalues of the Hubbard matrix are not strongly clustered, they only appear to be with respect to the LAPACK 3.1 criterion (4.3). The representation tree has an 'artificially long' chain of clusters within clusters. As discussed in Section 4.1, this situation is a nightmare as it requires repeated refinement of eigenvalues for each RRR with a deep representation tree. With respect to parallelism, this situation is equally critical as the depth of the representation tree becomes very uneven and thus the workload distribution extremely unbalanced between the processors. Since the representation tree unravels only during repeated eigenvalue refinement, such a case is not detectable in advance without significant work.

In [63], this 'spectrum peeling' problem of artificial clusters within clusters is analyzed and a solution proposed and evaluated. It consists of supplementing the threshold $\tau_{\text{minrgp}} = 10^{-3}$ with another criterion to limit possibly redundant eigenvalue refinement. The additional criterion is based on the *absolute* gap separation rather than the relative one. Given the spectral diameter of an (unreduced) tridiagonal

matrix, one can compute the average spectral (absolute) gap avgap := $\text{spdiam}/_{n-1}$. Then, the non-singletons $\hat{\lambda}_i, \hat{\lambda}_{i+1}$ are determined to belong to two different groups if

$$\text{rgap}(\hat{\lambda}_i) = |\hat{\lambda}_{i+1} - \hat{\lambda}_i| \geq \text{avgap}; \quad \hat{\lambda}_i, \hat{\lambda}_{i+1} \text{ no singletons.} \tag{4.4}$$

This criterion is similar to the one used in inverse iteration [39, 48] where the absolute distance between eigenvalues is used as indicator of whether Gram-Schmidt orthogonalization is necessary.

A proof of correctness legitimates the use of this new criterion in PDSYEVR. For the Hubbard matrix investigated above, the new combined criterion results in a maximum cluster size of 17, resulting in perfect scalability while achieving significantly better accuracy than the LAPACK 3.0 criterion. This and more experimental evidence is given in [63].

5. Performance analysis of pdsyevr. This section states some evaluation results obtained on our code. We show how the tridiagonal part of our new code, in particular with the enhancement discussed in Section 4.4, scales on larger numbers of processors. The results are compared with the tridiagonal part of ScaLAPACK's Divide & Conquer (D&C) [60].

As the discussion in [21, 22] shows, performance of sequential LAPACK eigensolvers can strongly depend on the matrix at hand, the same is true for the parallel case. For this reason, we are very grateful for help from collaborators who supply us with relevant test cases and whose feedback is invaluable for understanding design tradeoffs and practical challenges. We mention again [66] (an update of [67]) which compares a previous version of our code [4] with the ScaLAPACK eigensolvers PDSYEVX, PDSYEVD [60] and also ParEig [7] from PLAPACK; some results are also reported for the block Divide & Conquer algorithm [34, 6]. Another reference includes a pending update of [10]. Appendix D gives a short review of applications which can both supply such systems and benefit from our new code. The test matrices studied in this paper are summarized in Table 5.1.

Matrix name	Dimension	Origin
poly8	8000	B. Ward & Y. Bai, UTK
poly16	16000	B. Ward & Y. Bai, UTK
lapw	22908	A. Tate, CRAY
Poisson (1-2-1)	40001	generated
Wilkinson W^+_{2m+1}	40001	generated
Hubbard	63504	B. Ward & Y. Bai, UTK

TABLE 5.1

Tridiagonal test matrices used in this paper. Application matrices are from electronic structure calculations and computational quantum chemistry. 'Poisson' refers to a Toeplitz matrix with diagonal two and off-diagonals one. W^+_{2m+1} is the famous Wilkinson matrix.

Results shown in this paper were obtained on an IBM SP5 (Power 5, AIX), with 1.9 GHz and 7.6 Gflop/s peak, configured as SMP with 8 processors per node. The compile options `-O3 -qstrict -q64 -qarch=auto -qtune=auto` were used. Furthermore, Remote Direct Memory Access (RDMA) via the HPS (High Performance Switch, or "Federation") was enabled.

As test setup, we distributed the tridiagonal matrix to all processors and then inserted an MPI barrier in front of the calls to MPI_Wtime before and after the tridiagonal eigensolver. This guarantees that the reported runtime reflects potential load imbalances in the tridiagonal part.

In addition to giving runtime plots, we also report parallel efficiency. This should allow the reader to better judge scalability. The efficiency E for p processors is defined as the quotient

$$E(p) = \frac{t_{p_{\text{ref}}} * p_{\text{ref}}}{t_p * p}, \tag{5.1}$$

where t_p is the runtime for p processors and p_{ref} denotes the reference processor configuration, the smallest number of processors on which the problem was run.

To ensure load and memory balancing, we used Integrated Performance Monitoring (IPM,[37, 36]).

5.1. Scalability analysis of the tridiagonal part. Figures 5.1 and 5.2 compare MRRR and Divide & Conquer on poly8 and poly16. The poly8 matrix is almost too small to scale the codes up to 256 processors. One can see the efficiency drop noticeably beyond 128 processors. In the poly16 case, MRRR is slightly faster as seen in Figure 5.2.

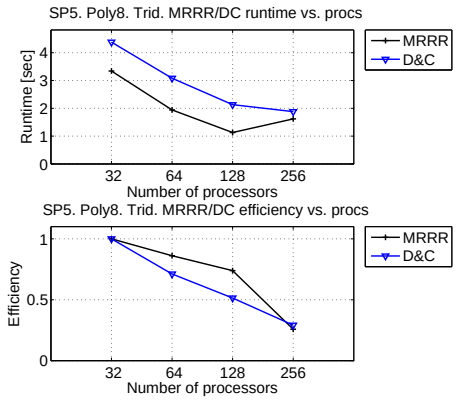

FIG. 5.1. *Poly8. Runtime and efficiency of the tridiagonal MRRR/D&C part on the IBM SP5.*

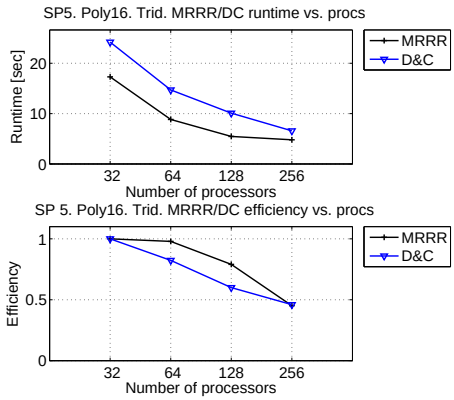

FIG. 5.2. *Poly16. Runtime and efficiency of the tridiagonal MRRR/D&C part on the IBM SP5.*

Figures 5.3 and 5.4 show the results for the LAPW and Hubbard matrices. The two algorithms behave comparably in Figure 5.3. However, Figure 5.4 is particularly noteworthy. It shows that indeed our code now scales on the Hubbard case. Thus, the worst-case behavior of our previous code [4] on this matrix that was observed in [66] is now remedied.

At last, we show two test matrices on which MRRR and D&C are known to vary widely. Figure 5.5 shows the results for the 1-2-1 matrix which is known to be difficult for Divide & Conquer. In this case, MRRR beats it hands down. Rather the opposite happens in the case of the Wilkinson matrix for which results are shown in Figure 5.6. It is known [21, 22] that here, D&C benefits enormously from deflation. On the other hand, the closeness of the eigenvalues requires substantial work by MRRR. This shows that one cannot expect one code to perform best in all circumstances. It will depend on the application which of the two algorithms is preferable.

5.2. Computing subsets of eigenpairs. This section gives a brief case study for the computation of subsets of eigenpairs. We evaluate the subset feature of our

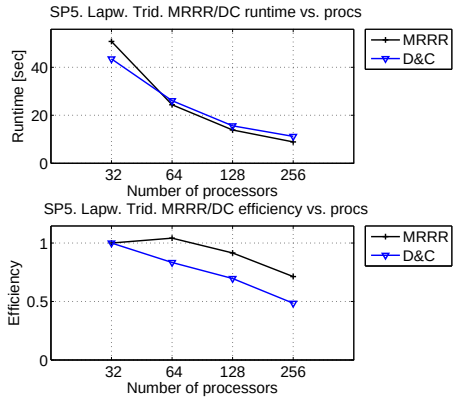

FIG. 5.3. *Lapw. Runtime and efficiency of the tridiagonal MRRR/D&C part on the IBM SP5.*

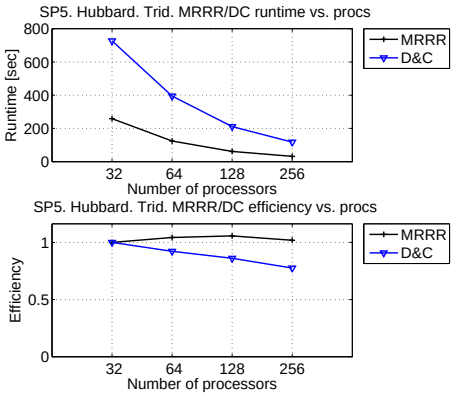

FIG. 5.4. *Hubbard. Runtime and efficiency of the tridiagonal MRRR/D&C part on the IBM SP5.*

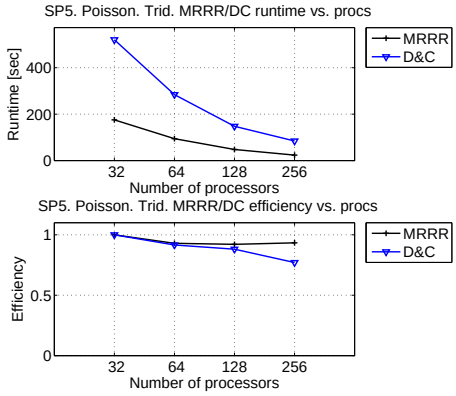

FIG. 5.5. *Poisson. Runtime and efficiency of the tridiagonal MRRR/D&C part on the IBM SP5.*

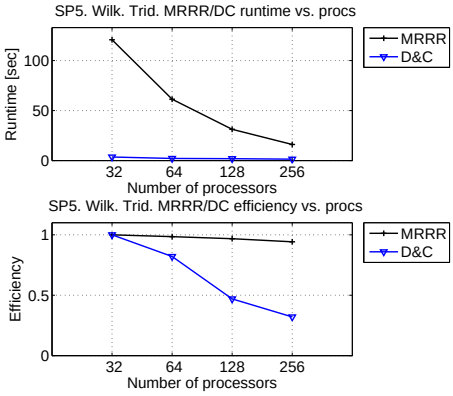

FIG. 5.6. *Wilkinson. Runtime and efficiency of the tridiagonal MRRR/D&C part on the IBM SP5.*

code on the task of computing 10% of the eigenvalues at the left end of the spectrum. ¶ There are three questions of interest.

1. What fraction of time does the parallel subset computation take compared to the full spectrum?
2. How does the subset computation scale?
3. What is the impact of the reduced cost of the tridiagonal part for subsets on the dense computation?

In order to answer the first question, subset efficiency can be computed relative to the runtime for the corresponding full spectrum computation. Let n denote the matrix dimension and m the number of eigenpairs in the subset, then we define subset efficiency as

$$S(p) = \frac{t_{(n,p)} * m}{t_{(m,p)} * n}, \tag{5.2}$$

¶For the importance of the subset feature in applications, see Appendix D.

where $t_{(i,p)}$ denotes the time taken for i eigenvalues on p processors.

As can be seen from the subset efficiency shown in Figure 5.7, definition (5.2) is slightly flawed in that for the subset considered, the efficiency is greater than one. This means that the representation tree for this subset (the leftmost 10% of the eigenpairs) involves less than 10% of the time spent on the representation tree for the full set of eigenpairs.

For the second question, by (5.1) efficiency can be computed using the subset runtime on the smallest feasible number of processors as reference. This is shown in Figure 5.8.

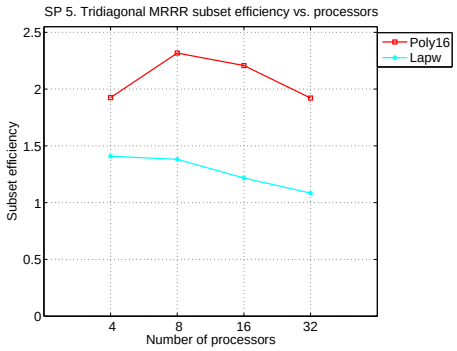

FIG. 5.7. *Subset efficiency relative to the full spectrum computation as defined by (5.2).*

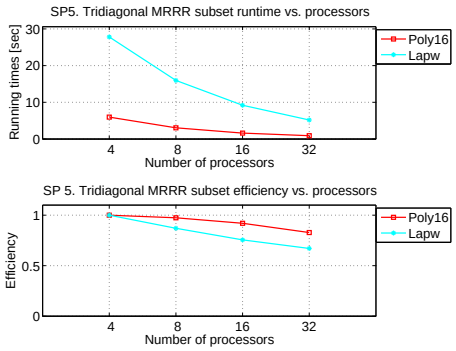

FIG. 5.8. *Efficiency of the tridiagonal subset computation for a fixed subset by (5.1).*

Computing subsets of eigenpairs instead of all of them is always beneficial because of the smaller amount of memory required. As seen in the example, a subset computation can also be easier than computing the full set of eigenpairs. In any event, the use of the subset feature influences the total time of the dense algorithm in two ways: first, the tridiagonal eigensolver takes less time and second, the backtransformation becomes cheaper. The reduction of the original dense matrix to tridiagonal form is unchanged and thus constitutes a lower bound for the fastest time possible for computing any part of the spectrum with a ScaLAPACK algorithm. For the matrices tested in [66], the reduction to tridiagonal form typically required between 50%-60% of the time for PDSYEVR and thus use of the subset feature can speed up the overall computation by at most a factor of about 1.7-2. The exact speedup depends on the percentage of eigenvalues required and also the particular matrix. This issue has to be investigated in the future in the context of the application at hand.

6. Conclusion and future work. In this paper, we described the design of ScaLAPACK's new MRRR algorithm. The novel features of this code are the use of static memory and a refined representation tree for achieving better parallel scalability even with stricter accuracy thresholds. The latter feature was shown to significantly improve load imbalance and scalability, curing some negative results from [66] on an earlier version [4].

Performance comparisons show that it is matrix and application dependent, which of the two principal algorithms, MRRR and Divide & Conquer, is the fastest algorithms for computing all eigenpairs. MRRR can compute subsets at reduced cost, using $\mathcal{O}(nk/p)$ operations per processor for the tridiagonal part. It remedies an issue of inverse iteration that does not guarantee a correct answer. The software is available from the authors on request and will be part of the next ScaLAPACK release.

Furthermore, we are studying an out-of-core extension and the impact of novel computer architectures on our code.

Acknowledgments. This work owes much to B. Parlett and I. Dhillon; it would not have been possible without their ground-breaking work on the MRRR algorithm. D. Antonelli contributed to the initial version of this code as part of his CS 267 class project in the spring and summer of 2005. We are very grateful to P. Bientinesi for many discussions and comments on the contents of this paper. We also thank J. Demmel and J. Langou for their comments, and the referees for their critical reading of this manuscript that pushed the code development as well. The editors encouraged us to pursue work on this paper, a long email exchange with J. Reid contributed significantly to improving the presentation. We thank B. Ward, Y. Bai, A. Sunderland, and E. Apra for evaluating several versions of our software and for their very useful feedback. Thanks also to who A. Tate and J. Lewis who evaluated our code and contributed some early benchmark results and feedback. Lastly, we gratefully acknowledge the use of the IBM SP3 seaborg.nersc.gov and SP5 bassi.nersc.gov that we used for development and testing.

Appendix A. Testing.

A.1. Design of the tester. As part of the current release, we provide a tester for PDSYEVR. The tester allows the user to specify matrix sizes, matrix types, and processor configurations. There are six different tests available: The user can choose whether to compute eigenpairs or eigenvalues only; furthermore (s)he can choose between the full spectrum, a range of eigenpairs specified by index, or the eigenpairs from an interval.

The tester verifies that the computed eigenpairs have small residuals, that is they satisfy

$$\|(A - \lambda I)z\| = \mathcal{O}(\epsilon \|A\|). \tag{A.1}$$

We use ScaLAPACK's PDSEPCHK for this test. Furthermore, the computed eigenvectors must be numerically orthogonal, satisfying

$$\|I - Z^T Z\| = \mathcal{O}(n\epsilon). \tag{A.2}$$

ScaLAPACK's PDSEPQTQ is used for this test. By choosing a threshold in either of these tests, the user can decide what is considered a failure. Typically, the derivation from orthogonality is below $100n\epsilon$.

For subset tests, the code also checks that the computed eigenvalues are consistent with those computed for the full-spectrum test. Lastly, the tester performs memory consistency checks to ensure that no memory is accessed outside the assigned workspace.

The driver PDSEPRDRIVER is the main program that initializes the ScaLAPACK environment. The processor 0 is responsible for I/O, it reads in the current test from the file 'SEPR.dat' and prints a diagnostic message at the end of the test. PDSEPRREQ partitions the available memory appropriately for the processor configuration and matrix at hand and passes it to the test subroutine PDSEPRTST.

A.2. Test matrices. We use the test matrices from the ScaLAPACK test matrix collection for the symmetric eigenvalue problem, see also the ScaLAPACK Installation Guide [13].

1. The zero matrix

2. The identity matrix
3. A diagonal matrix with uniformly spaced entries from 1 to ϵ and random signs
4. A diagonal matrix with geometrically spaced entries from 1 to ϵ and random signs
5. A diagonal matrix with entries 1, ϵ, ϵ, ..., ϵ and random sign
6. Same as (4), but multiplied by SQRT(overflow threshold)
7. Same as (4), but multiplied by SQRT(underflow threshold)
8. Same as (3), but pre- and post-multiplied by an orthogonal matrix and its transpose, respectively
9. Same as (4), but pre- and post-multiplied by an orthogonal matrix and its transpose, respectively
10. Same as (5), but pre- and post-multiplied by an orthogonal matrix and its transpose, respectively
11. Same as (8), but multiplied by SQRT(overflow threshold)
12. Same as (8), but multiplied by SQRT(underflow threshold)
13. A symmetric matrix with random entries chosen uniformly from (-1, 1)
14. Same as (13), but multiplied by SQRT(overflow threshold)
15. Same as (13), but multiplied by SQRT(underflow threshold)
16. Same as (8), but diagonal elements are all positive.
17. Same as (9), but diagonal elements are all positive.
18. Same as (10), but diagonal elements are all positive.
19. Same as (16), but multiplied by SQRT(overflow threshold)
20. Same as (16), but multiplied by SQRT(underflow threshold)
21. A tridiagonal matrix that is a direct sum of smaller diagonally dominant submatrices. Each unreduced submatrix has geometrically spaced diagonal entries from 1 to ϵ.
22. A matrix of the form $U'DU$, where U is orthogonal and D has $\lceil \lg n \rceil$ "clusters" at $0, 1, 2, \ldots, \lceil \lg n \rceil - 1$. The size of the cluster at the value I is 2^I.

A.3. A note on the current ScaLAPACK orthogonality test. There is a subtle flaw in the current ScaLAPACK tester for PDSYEVX. The tester does *not* verify that (A.2) holds. It only verifies (A.2) for those eigenpairs that PDSYEVX could re-orthogonalize with the amount of memory supplied. This is not transparent to the user, unfortunately.

An inspection of the orthogonality test PDSEPQTQ reveals that the tester scales the submatrix of $C = I - Z^T Z$ belonging to unresolved clusters by a quantity 1.0D2 $*$ gap. Thus, PDSEPQTQ misrepresents the orthogonality. Computed eigenvectors with poor orthogonality are *not* reported as failure when the corresponding eigenvalues belong to tight clusters (that is, they have small gaps), and full reorthogonalization was impossible within the available amount of memory.

When we substituted PDSEPQTQ by a 'true' orthogonality test, we found that PDSYEVX fails on two examples in the current ScaLAPACK tester. Both these failures are on matrices of type 10, we do not report them here. Our algorithm PDSYEVR does not fail.

Appendix B. Two illustrative examples.

We provide here an example for a failure of the subset-based parallelization and show that the embedded approach succeeds.

B.1. How subset-based parallelization can fail. This section gives a short example to illustrate how the simple approach discussed in Section 3.2.1, the parallelization based on the subset feature of the sequential code, can fail.

Our example is the matrix given in (B.1). It stems from the test set described in Appendix A.2 and is of type 10.

$$\mathbf{D} = \begin{bmatrix} 7.603714754255181 \times 10^{-2} \\ 9.239628524574373 \times 10^{-1} \\ 1.090348352668058 \times 10^{-14} \\ -1.103610618742564 \times 10^{-14} \\ -1.110886780132261 \times 10^{-14} \end{bmatrix}, \mathbf{E} = \begin{bmatrix} 2.650575404250068 \times 10^{-1} \\ 6.006645802706129 \times 10^{-15} \\ -9.033212524378735 \times 10^{-16} \\ 1.373587734806744 \times 10^{-17} \end{bmatrix} \tag{B.1}$$

Its eigenvalues are shown in (B.2).

$$\mathbf{W} = \begin{bmatrix} -1.113099956921839 \times 10^{-14} \\ -1.110886780132261 \times 10^{-14} \\ -1.099358132331527 \times 10^{-14} \\ 1.110223024625157 \times 10^{-14} \\ 1 \end{bmatrix} \tag{B.2}$$

The experiment used $p = 2$ processors; eigenpairs $1 - 3$ are assigned to the first and eigenpairs $4, 5$ are assigned to the second processor. Each processor calls the sequential code (LAPACK's DSTEGR) on its subset. The crossproduct of the computed eigenvector matrix is shown in (B.3).

$$\mathbf{Z}' \cdot \mathbf{Z} = \left[\begin{array}{ccc|cc} 1 & 0 & -1.1 \times 10^{-16} & 1.3 \times 10^{\mathbf{-4}} & -1.2 \times 10^{-17} \\ 0 & 1 & 0 & 0 & 0 \\ -1.1 \times 10^{-16} & 0 & 1 & -7.5 \times 10^{\mathbf{-4}} & -1.8 \times 10^{-17} \\ \hline 1.3 \times 10^{\mathbf{-4}} & 0 & -7.5 \times 10^{\mathbf{-4}} & 1 & 3.7 \times 10^{-18} \\ -1.2 \times 10^{-17} & 0 & -1.8 \times 10^{-17} & 3.7 \times 10^{-18} & 1 \end{array}\right] \tag{B.3}$$

Within the subset assigned to each processor, the orthogonality is fine, see the diagonal blocks in (B.3). However, the eigenvectors computed by the first processor are not orthogonal to the fourth eigenvector computed by the other processor.

B.2. Test result for the conformal embedding. For the same processor configuration as in the previous section, and the sample matrix (B.1), we show in (B.4) the crossproduct of the eigenvectors computed with the embedded approach from Section 3.2.2. This time, the eigenvector matrix is numerically orthogonal as desired.

$$\mathbf{Z}' \cdot \mathbf{Z} = \left[\begin{array}{ccc|cc} 1 & 0 & 1.1 \times 10^{-15} & 2.1 \times 10^{-17} & -1.1 \times 10^{-17} \\ 0 & 1 & 0 & 0 & 0 \\ 1.1 \times 10^{-15} & 0 & 1 & 1.7 \times 10^{-17} & -4.7 \times 10^{-17} \\ \hline 2.1 \times 10^{-17} & 0 & 1.7 \times 10^{-17} & 1 & 3.4 \times 10^{-19} \\ -1.1 \times 10^{-17} & 0 & -4.7 \times 10^{-17} & 3.4 \times 10^{-19} & 1 \end{array}\right] \tag{B.4}$$

Appendix C. The interface of pdsyevr.

This section compares the interfaces of PDSYEVR and PDSYEVX. For brevity, we only point out the differences between the two interfaces. The most important difference, namely the memory requirements, have already been addressed in Section 3.3.

For reference, we first show the interface of PDSYEVX from ScaLAPACK version 1.7. The meaning of all arguments is documented in [14].

```
      SUBROUTINE PDSYEVX( JOBZ, RANGE, UPLO, N, A, IA, JA, DESCA, VL,
     $                    VU, IL, IU, ABSTOL, M, NZ, W, ORFAC, Z, IZ,
     $                    JZ, DESCZ, WORK, LWORK, IWORK, LIWORK, IFAIL,
     $                    ICLUSTR, GAP, INFO )
```

In comparison, we show the corresponding interface of PDSYEVR.

```
      SUBROUTINE PDSYEVR( JOBZ, RANGE, UPLO, N, A, IA, JA, DESCA, VL,
     $                    VU, IL, IU, M, NZ, W, Z, IZ,
     $                    JZ, DESCZ, WORK, LWORK, IWORK, LIWORK,
     $                    INFO )
```

Common arguments generally have the same type and meaning as for PDSYEVX. However, five parameters have been suppressed. The parameters ABSTOL and ORFAC are related to bisection and inverse iteration and have no meaning in MRRR. The parameters IFAIL, ICLUSTR, and GAP were used to report clusters for which re-orthogonalization could not be applied due to insufficient memory. MRRR avoids re-orthogonalization entirely, thus these parameters are no longer needed.

Appendix D. Impact on applications.

This section aims to gauge the impact of our work on applications, specifically the field of electronic structure computations which makes heavy use of dense eigensolvers. Many of these calculations are based on the solution of effective single particle Schrödinger equations due to Kohn and Sham [42] which are eigenvalue problems. The choice of the most suitable method for their solution is determined by the discretization basis for the problem and the number of eigenvectors ('states') that have to be computed.

In a plane wave (PW) basis where the matrix ('the Hamiltonian') is usually only available implicitly through matrix-vector multiplication, iterative methods such as Nonlinear Conjugate Gradient [53, 64], Lanczos [48], or Jacobi-Davidson [58] are preferred. Examples include ESCAN [11], PARATEC [55], and VASP [43]. Real space discretization of the Hamiltonian as for example used in PARSEC [12, 1] generally lead to a large sparse matrix so that again iterative eigenvalue methods are employed. See [56] for a more detailed review.

On the other hand, direct eigensolvers as provided by ScaLAPACK [14] are used whenever a part or all of the spectrum of a dense matrix, at least 10% say, needs to be computed. Note that eigenvalue subsets at the left end of the spectrum usually describe the states that are occupied by the electrons of the system. One example is NWChem [40, 5] which is based on Gaussian basis discretization and typically computes the full spectrum. Another example is multi-band $k \cdot p$ computation relying on direct diagonalization [61]. Linearized-Augmented-Plane-Wave (LAPW) codes like WIEN2k [57] use a hybrid approach where a plane wave basis is used outside the atomic region which is discretized using spherical harmonics. Lastly, the Linear Combination of Bulk Bands (LCBB) method [65] uses a direct eigensolver.

Second and of equally high importance, all the iterative methods mentioned above use a direct eigensolver for Rayleigh-Ritz subspace diagonalization. Methods which work with larger subspaces, for example Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) [41], are particularly dependent on this part being scalable.

We quote as an example from a recent review on numerical methods for electronic structure calculations [56]: 'Interestingly, the large and dense eigenvalue problem will gain importance as systems become larger. This is because most methods solve a dense eigenvalue problem which arises from projecting the Hamiltonian into some subspace. As the number of states increases, this dense problem can reach sizes in the tens of thousands. Because of the cubic scaling of standard eigenvalue methods for dense matrices, these calculations may become a bottleneck.'

REFERENCES

[1] M. M. G. Alemany, M. Jain, L Kronik, and J. R. Chelikowsky. Real-space pseudopotential method for computing the electronic properties of periodic systems. *Phys. Rev. B*, 69:075101, 2004.

[2] P. Alpatov, G. Baker, C. Edwards, J. Gunnels, G. Morrow, J. Overfelt, R. A. van de Geijn, and J. Y.-J. Wu. PLAPACK: parallel linear algebra package design overview. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–16, New York, NY, USA, 1997. ACM Press.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK User's Guide*. SIAM, Philadelphia, 3. edition, 1999.

[4] D. Antonelli and C. Vömel. LAPACK working note 168: PDSYEVR. ScaLAPACK's parallel MRRR algorithm for the symmetric eigenvalue problem. Technical Report UCBCSD-05-1399, University of California, Berkeley, 2005.

[5] E. Apra et al. NWChem, a computational chemistry package for parallel computers, version 4.7. Technical report, Pacific Northwest National Laboratory, Richland, WA. USA, 2005.

[6] Y. Bai and R. C. Ward. A Parallel Symmetric Block-Tridiagonal Divide-and-Conquer Algorithm. *ACM Trans. Math. Software*, 33(4):A25, to appear, 2007. Revised version of UT-CS-05-571, University of Tennessee.

[7] P. Bientinesi, I. S. Dhillon, and R. A. van de Geijn. A Parallel Eigensolver for Dense Symmetric Matrices based on Multiple Relatively Robust Representations. *SIAM J. Sci. Comput.*, 27(1):43–66, 2005.

[8] L. S. Blackford, A. Cleary, A. Petitet, R. C. Whaley, J. Demmel, I. Dhillon, H. Ren, K. Stanley, J. J. Dongarra, and S. Hammarling. Practical experience in the numerical dangers of heterogeneous computing. *ACM Trans. Math. Software*, 23(2):133–147, 1997.

[9] L. S. Blackford, J. W. Demmel, J. J. Dongarra, I. S. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Software*, 28(2):135–151, 2002.

[10] E. Breitmoser and A. G. Sunderland. A performance study of the PLAPACK and ScaLAPACK Eigensolvers on HPCx for the standard problem. Technical Report HPCxTR0406, The HPCx Consortium, 2004.

[11] A. Canning, L.-W. Wang, A. Williamson, and A. Zunger. Parallel Empirical Pseudopotential Electronic Structure Calculations for Million Atom Systems. *J. Comp. Phys.*, 160:29–41, 2000.

[12] J. R. Chelikowsky, N. Troullier, and Y. Saad. The finite-difference-pseudopotential method: Electronic structure calculations without a basis. *Phys. Rev. Lett.*, 72:1240–1243, 1994.

[13] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Installation guide for ScaLAPACK. Computer Science Dept. Technical Report CS-95-280, University of Tennessee, Knoxville, TN, 1995. (Also LAPACK Working Note #93).

[14] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97:1–15, 1996. (also as LAPACK Working Note #95).

[15] J. Choi, J. Dongarra, and D. Walker. The Design of a Parallel Dense Linear Algebra Software Library: Reduction to Hessenberg, Tridiagonal, and Bidiagonal Form. Technical Report UT-CS-95-275, University of Tennessee, Knoxville, TN, USA, 1995. (LAPACK Working Note #80).

[16] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, and D. Walker. A proposal for a set of parallel basic linear algebra subprograms. Technical Report UT-CS-95-292, University of

Tennessee, Knoxville, TN, USA, 1995. (LAPACK Working Note #100).
[17] C. Davis and W. Kahan. The rotation of eigenvectors by a perturbation. III. *SIAM J. Numer. Anal.*, 7(1):1–47, 1970.
[18] J. W. Demmel, I. S. 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, 1995. (LAPACK working note #70).
[19] J. W. Demmel and J. J. Dongarra. LAPACK 2005 Prospectus: Reliable and Scalable Software for Linear Algebra Computations on High End Computers. Technical Report UT-CS-05-546, University of Tennessee, Knoxville, TN, USA, 2005. (LAPACK Working Note #164).
[20] J. W. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 11(5):873–912, 1990.
[21] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel. A Testing Infrastructure for Symmetric Tridiagonal Eigensolvers. Technical report LBNL-61831, Lawrence Berkeley National Laboratory, 2006.
[22] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel. Lapack Working Note 183: Performance and Accuracy of LAPACK's Symmetric Tridiagonal Eigensolvers. Technical report, UC Berkeley, 2006.
[23] J. W. 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 Processing for Scientific Computing, San Francisco*, Philadelphia, 1994. SIAM. (also LAPACK working note #86).
[24] I. S. Dhillon. *A New $O(n^2)$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*. PhD thesis, University of California, Berkeley, California, 1997.
[25] I. S. Dhillon. Current inverse iteration software can fail. *BIT*, 38:4:685–704, 1998.
[26] I. S. Dhillon and B. N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and Appl.*, 387:1–28, 2004.
[27] I. S. Dhillon and B. N. Parlett. Orthogonal eigenvectors and relative gaps. *SIAM J. Matrix Anal. Appl.*, 25(3):858–899, 2004.
[28] I. S. Dhillon, B. N. Parlett, and C. Vömel. Glued matrices and the MRRR algorithm. *SIAM J. Sci. Comput.*, 27(2):496–510, 2005. Revised version of LAPACK Working Note 163.
[29] I. S. Dhillon, B. N. Parlett, and C. Vömel. The design and implementation of the MRRR algorithm. *ACM Trans. Math. Software*, 32(4):533–560, 2006.
[30] J. Dongarra and R. C. Whaley. A User's Guide to the BLACS v1.1. Technical Report UT-CS-95-281, University of Tennessee, Knoxville, TN, USA, 1995. (LAPACK Working Note #94).
[31] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16:1–17, 1990.
[32] J. J. Dongarra, J. J. Du Croz, S. Hammarling, and R. J. Hanson. An extented set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–17, 1988.
[33] J. J. Dongarra and R. A. van de Geijn. Lapack working note 37: Two dimensional basic linear algebra communication subprograms. Technical Report UT-CS-91-138, University of Tennessee, Knoxville, TN, USA, 1991.
[34] W. N. Gansterer, R. C. Ward, R. P. Muller, and W. A. Goddard III. Computing Approximate Eigenpairs of Symmetric Block Tridiagonal Matrices. *SIAM J. Sci. Comput.*, 25(1):65–85, 2003.
[35] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, 1994.
[36] Integrated Performance Monitoring (IPM). http://www.nersc.gov/nusers/resources/ software/tools/ipm.php, 2007.
[37] IPM: Integrated Performance Monitoring. http://ipm-hpc.sourceforge.net, 2007.
[38] I. C. F. Ipsen. A history of inverse iteration. In B. Huppert and H. Schneider, editors, *Helmut Wielandt, Mathematische Werke, Mathematical Works*, volume II: Matrix Theory and Analysis. Walter de Gruyter, Berlin, 1996.
[39] I. C. F. Ipsen. Computing an eigenvector with inverse iteration. *SIAM Review*, 39(2):254–291, 1997.
[40] R. A. Kendall et al. High Performance Computational Chemistry: An overview of NWChem a distributed parallel application. *Computer Phys. Comm.*, 128:260–283, 2000.
[41] A. V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.*, 23(2):517–541, 2001.
[42] W. Kohn and L. S. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140A:1133–1140, 1965.
[43] G. Kresse and J. Furthmüller. Efficiency of ab-initio total energy calculations for metals and

semiconductors using a plane-wave basis set. *Comp. Mater. Sci.*, 6:15–50, 1996.
[44] Lapack 3.1. http://www.netlib.org/lapack/lapack-3.1.0.changes, 2006.
[45] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Software*, 5:308–323, 1979.
[46] O. A. Marques, B. N. Parlett, and C. Vömel. LAPACK working note 167: Subset computations with the MRRR algorithm. Technical Report UCBCSD-05-1392, University of California, Berkeley, 2005.
[47] B. N. Parlett. *Acta Numerica*, chapter The new qd algorithms, pages 459–491. Cambridge University Press, Cambridge, UK, 1995.
[48] B. N. Parlett. *The Symmetric Eigenvalue Problem*. SIAM Press, Philadelphia, PA, 1998.
[49] B. N. Parlett. For tridiagonals T replace T with LDL^t. *J. Comp. Appl. Math.*, 123:117–130, 2000.
[50] B. N. Parlett and I. S. Dhillon. Fernando's solution to Wilkinson's problem: an application of double factorization. *Linear Algebra and Appl.*, 267:247–279, 1997.
[51] B. N. Parlett and I. S. Dhillon. Relatively robust representations of symmetric tridiagonals. *Linear Algebra and Appl.*, 309(1-3):121–151, 2000.
[52] B. N. Parlett and O. Marques. An implementation of the dqds algorithm (positive case). *Linear Algebra and Appl.*, 309:217–259, 2000.
[53] M. C. Payne, M. P. Teter, D. C. Allan, T. A. Arias, and J. D. Joannopoulos. Iterative minimization techniques for ab initio total-energy calculations: Molecular dynamics and conjugate gradients. *Rev. Mod. Phys.*, 64(4):1045–1097, 1992.
[54] A. Petitet, H. Casanova, J. J. Dongarra, Y. Robert, and R. C. Whaley. Parallel and Distributed Scientific Computing: A Numerical Linear Algebra Problem Solving Environment Designer's Perspective. In B. Plateau J. Blazewicz, K. Ecker and D. Trystram, editors, *Handbook on Parallel and Distributed Processing, International Handbook on Information Systems*, volume 3. Springer, 2000. (also as LAPACK Working Note #139).
[55] B Pfrommer, J. Demmel, and H. Simon. Unconstrained energy functionals for electronic structure calculations. *J. Comp. Phys.*, 150:287, 1999.
[56] Y. Saad, J. R. Chelikowsky, and S. M. Shontz. Numerical methods for electronic structure calculations of materials. Technical Report umsi-2006-15, University of Minnesota, Department of Computer Science and Engineering, March 2006.
[57] D. Sanchez-Portal, P. Ordejon, Artacho E., and J. M. Soler. Density functional method for very large systems with lcao basis sets. *Int. J. Quant. Chem.*, 65:453–461, 1997.
[58] G. L. G. Sleijpen and H. A. Van der Vorst. A Jacobi-Davidson iteration method for linear eigenvalue problems. *SIAM J. Matrix Anal. Appl.*, 17(2):401–425, 1996.
[59] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1996.
[60] F. Tisseur and J. Dongarra. A Parallel Divide and Conquer algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. Comput.*, 6(20):2223–2236, 1999.
[61] S. Tomic, A. G. Sunderland, and I. J. Bush. Parallel multi-band $k\dot{p}$ code for electronic structure of zinc blend semiconductor quantum dots. *J. Mater. Chem.*, 16:1963–1972, 2006.
[62] R. A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. MIT Press, Cambridge, MA, USA, 1997.
[63] C. Vömel. LAPACK Working Note 194. A refined representation tree for MRRR. Technical report, University of Tennessee, Knoxville, TN, USA, 2007.
[64] L.-W. Wang and A. Zunger. Solving Schrödinger's equation around a desired energy: Application to silicon quantum dots. *J. Chem. Phys.*, 100:2394–2397, 1994.
[65] L.-W. Wang and A. Zunger. Linear combination of bulk band method for strained system million atom nanostructure calculations. *Phys. Rev. B*, 59:15806, 1999.
[66] R. C. Ward and Y. Bai. Performance of Parallel Eigensolvers on Electronics Structure Calculations II. Technical Report UT-CS-06-572, University of Tennessee, Knoxville, TN, USA, 2006.
[67] R. C. Ward, Y. Bai, and J. Pratt. Performance of Parallel Eigensolvers on Electronics Structure Calculations. Technical Report UT-CS-05-560, University of Tennessee, Knoxville, TN, USA, 2005.