LAPACK WORKING NOTE 168: PDSYEVR. SCALAPACK'S PARALLEL MRRR ALGORITHM FOR THE SYMMETRIC EIGENVALUE PROBLEM[§]

DOMINIC ANTONELLI[‡] AND CHRISTOF VÖMEL[‡]

Technical Report UCB//CSD-05-1399
Computer Science Division
UC Berkeley

Abstract. In the 90s, Dhillon and Parlett devised a new algorithm (Multiple Relatively Robust Representations, MRRR) for computing numerically orthogonal eigenvectors of a symmetric tridiagonal matrix T with $\mathcal{O}(n^2)$ cost. In this paper, we describe the design of PDSYEVR, a ScaLAPACK implementation of the MRRR algorithm to compute the eigenpairs in parallel. It represents a substantial improvement over the symmetric eigensolver PDSYEVX that is currently in ScaLAPACK and is going to be part of the next ScaLAPACK release.

AMS subject classifications. 65F15, 65Y15.

Key words. Multiple relatively robust representations, ScaLAPACK, symmetric eigenvalue problem, parallel computation, numerical software, design, implementation.

1. Introduction. Starting in 2005, the National Science Foundation is funding an initiative [10] to improve the LAPACK [2] and ScaLAPACK [7] 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.

The NSF proposal [10] identifies the parallelization of the MRRR algorithm [22, 23, 24, 25, 26, 13] as one key improvement to ScaLAPACK. Specifically, the authors point out that in ScaLAPACK, there are 'parallel versions of only the oldest LAPACK algorithms for the symmetric eigenvalue decomposition.'

ScaLAPACK contains a parallel bisection code, PDSTEBZ, to compute eigenvalues of a symmetric tridiagonal matrix. These eigenvalues can be used to compute the corresponding eigenvectors by parallel inverse iteration PDSTEIN. For computing k eigenvalues of an $n \times n$ matrix, bisection is an $\mathcal{O}(kn)$ process. If the k corresponding eigenvectors are to be computed by bisection, then inverse iteration with reorthogonalization takes $\mathcal{O}(k^2 n)$ operations.

ScaLAPACK also provides an expert driver, PDSYEVX, that takes a dense distributed matrix, transforms it to tridiagonal form, computes the eigenvalues by PDSTEBZ and the eigenvectors by PDSTEIN, and then applies the appropriate orthogonal transformations to obtain the eigenvectors of the dense matrix. Both the reduction to symmetric tridiagonal form and the orthogonal transformation to obtain the eigenvectors of the initial matrix have $\mathcal{O}(n^3)$ complexity and require $\mathcal{O}(n^2)$ memory.

Thus, if perfect scalability among p processors could be achieved, then each processor would perform $\mathcal{O}(n^3/p)$ operations using $\mathcal{O}(n^2/p)$ memory. With respect to these numbers, parallelizing the MRRR algorithm that has $\mathcal{O}(n^2)$ complexity for the full tridiagonal eigenproblem seems less of a priority. However, there is one important

[‡]Computer Science Division, University of California, Berkeley, CA 94720, USA. {dantonel,voemel}@cs.berkeley.edu

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

reason for its parallelization. Memory scalability with $\mathcal{O}(n^2/p)$ memory per processor can only be achieved if MRRR is parallelized.

A parallel version, ParEig, of the MRRR algorithm for the tridiagonal eigenvalue problem has already been developed [3]. It is implemented in C and makes use of LAPACK 3.0 f77 kernels for the sequential code DSTEGR. Memory is allocated dynamically as needed; MPI [30] is used for parallel communication. The algorithm depends on PLAPACK [1, 31] for the orthogonal transformation to tridiagonal form and the backtransformation to obtain the eigenvectors of the original matrix.

We parallelized MRRR in the ScaLAPACK environment. Together with the accompanying tester, our algorithm is going to be part of the next ScaLAPACK release, making it available across a large number of platforms and to a significant user base. Our software was developed independently from [3]. Algorithmically, this work is excellent and there was nothing for us to improve on. However, by its policy, ScaLAPACK only uses memory provided by the user. This required us to find a way to implement the MRRR parallelization in fixed, minimal workspace. This is our scientific contribution.

It is important to note that our code substantially improves the functionality currently available in the ScaLAPACK library. PDSYEVX does *not* guarantee a correct answer with $\mathcal{O}(n^2/p)$ memory per processor. Depending on the spectrum of the matrix at hand, tridiagonal inverse iteration can require $\mathcal{O}(n^3)$ operations and $\mathcal{O}(n^2)$ memory *on a single processor* to guarantee the correctness of the computed eigenpairs in the worst case. If the eigenvalues of the matrix are tightly clustered, the eigenvectors have to be computed (and reorthogonalized against each other) on the same processor; parallelism in the inverse iteration is lost.

The rest of this paper is organized as follows. The general ScaLAPACK approach to parallel computation is described in Section 2.1. In Section 2.2, we describe the design of the new MRRR-based ScaLAPACK driver. In Section 3.1, we give a short introduction to the MRRR algorithm; we concentrate on those aspects that are necessary for understanding the challenges of its parallelization. In Section 3.2, we discuss various parallelization strategies with their benefits and shortcomings. Then we present our conclusions and ideas for future work. In the Appendix A, we describe the design of the ScaLAPACK tester for our code. Appendix C contains a description of the calling sequence and all relevant parameters.

2. Outline of the parallel design.

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, we refer to [29, 7].

2.1.1. ScaLAPACK, BLACS, and PBLAS. Except for the (significant) additional complexity of parallel communication, ScaLAPACK [7] algorithms look similar to their LAPACK [2] 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 [8], that are distributed-memory versions of the BLAS [5, 15, 16, 20].
- The Basic Linear Algebra Communication Subprograms, BLACS [14, 17], that 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, we look at Algorithm 1 that 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. First, the eigenvalues are computed by bisection [11]. Afterwards, the corresponding eigenvectors are computed by inverse iteration [12, 18, 19].

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

Input: distributed symmetric matrix A
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 each processor.
(3) Compute eigenvalues and eigenvectors in parallel (PDSTEBZ and PDSTEIN).
(4) Apply distributed orthogonal matrix Q to eigenvectors to obtain the distributed Z containing the eigenvectors of A (PDORMTR).

2.1.2. Memory and communication management. Both LAPACK and ScaLAPACK only use memory provided by the user, no memory is allocated internally. The required amount of workspace is an explicit part of the interface.

This policy can provide severe constraints for the developer. Firstly, algorithmic design requires careful and tedious workspace reuse and is prone to errors that are hard to debug. Secondly, for backward compatibility, interfaces should not be changed. Thus, algorithmic changes that require a different amount of workspace imply new interfaces that need to be propagated throughout the library.

On the other hand, this policy guarantees to the user that no LAPACK or ScaLAPACK library function aborts inexpectedly due to insufficient memory when dynamic allocation fails. Secondly, there is no unknown run-time cost in terms of memory and time on each allocation and de-allocation.

2.1.3. Communication management. The BLACS provide only *synchronous* send and receive routines for communication [7]. 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. In this section, we describe the general structure of PDSYEVR. We avoid here the specifics of the parallel eigencomputation and only focus on the correct use of the available ScaLAPACK routines.

The design of Algorithm 2 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.

The key steps are those from (2) to (4b). The tridiagonal matrix is shared across all processors in (2). Based on some division of work, the processors compute the wanted eigenvalues and eigenvectors in parallel. There is an optional communication between the eigenvalue (3a) and the eigenvector (3b) part which we discuss later. Then the eigenvalues are shared (4a), and the eigenvector matrix is distributed (4b).

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

Input: distributed symmetric matrix A
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 each processor.
(3a, 3b) Compute eigenvalues and eigenvectors in parallel using an MRRR-based kernel.
(4a) Share the computed eigenvalues across all processors.
(4b) Distribute the eigenvectors from local workspace into distributed storage for Z matrix(PDLAEVSWP).
(5) Apply distributed orthogonal matrix Q to eigenvectors to obtain the distributed Z containing the eigenvectors of A (PDORMTR).

We point out that we store Z using ScaLAPACK's 2D-block-cyclic layout, mainly to make the orthogonal matrix-matrix multiply in step 5 efficient. However, the computation of an eigenvector is done 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, i.e steps (3a) & (3b) from Algorithm 2 in Section 2.2. To set the stage, Section 3.1 gives a short overview of the MRRR algorithm. Afterwards, in Section 3.2, we discuss different ways of using it in a parallel environment.

3.1. A brief overview of the MRRR algorithm. This section presents a very short introduction to the MRRR algorithm in order to show different sources of parallelism in the computation. We assume that the reader is familiar with at least the general ideas of MRRR, for more details we refer to [13, 23, 24, 25, 26].

The ultimate goal of MRRR is to find a suitable (shifted) LDL^T factorization so that a local eigenvalue of interest has a large relative gap[1]. We denote the eigenvalue approximation by $\hat{\lambda}$ and require that

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

In this case, MRRR computes the corresponding computed eigenvector v in a way that guarantees v to have a small relative residual

$$\|(LDL^T - \hat{\lambda}I)v\| = \mathcal{O}(n\epsilon|\lambda|). \tag{3.2}$$

Then, the classical gap theorem [9, 21] implies

$$|\sin \angle(v, z)| \leq \frac{\|(LDL^T - \hat{\lambda}I)v\|}{\text{gap}(\hat{\lambda})} \equiv \frac{\mathcal{O}(n\epsilon)}{\text{relgap}(\hat{\lambda})}. \tag{3.3}$$

where z denotes the true eigenvector.

[1] We define 'suitable' by requiring that small relative changes in entries of L and D cause small relative changes in the eigenvalue. We call such an LDL^T a Relatively Robust Representation (RRR) for this eigenvalue.

In order to find such an RRR for each eigenvalue, the algorithm builds a *representation tree* [25]. Each node of the graph represents the RRR for a group of eigenvalues. The root node of this *representation tree* is the initial representation that is an RRR for all the wanted eigenvalues, each leaf corresponds to a *singleton*, that is a (shifted) eigenvalue with a large relative gap. The root node is derived directly from a shifted LDL^T factorization of the original tridiagonal matrix T. All subsequent RRRs are computed from the RRR of their parent node in the representation tree using the *differential stationary qd algorithm*. It provides a stable way of computing a child RRR (L^+, D^+) from its parent (L, D) via

$$(LDL^T - \sigma I) = L^+ D^+ (L^+)^T. \tag{3.4}$$

We summarize the MRRR procedure in Algorithm 3.

Algorithm 3 Principle outline of the (sequential) MRRR algorithm.

(1) For each unreduced block of T, compute a root representation $LDL^T = T - \sigma I$ and its wanted eigenvalues.
(2) Build the representation tree. Using a series of shifts and qd transformations of the form (3.4), find suitable RRRs so that, eventually, all eigenvalues become singletons.
(3) For each singleton compute the corresponding eigenvector so that (3.2) holds.

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

1. The computation of each root representation costs $\mathcal{O}(n)$ operations and is essentially sequential. However, the computation of eigenvalues by bisection is an embarrassingly parallel operation: each individual eigenvalue can be computed independently from all the others.
2. The computation of a child RRR from its parent is again essentially sequential. However, in order to detect large relative gaps in an RRR at hand, eigenvalues need to be refined by bisection. This is again embarrassingly parallel.
3. 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. A straightforward idea for parallelization is to use the subset feature of the sequential code (Algorithm 3). (See [28] for a discussion of how the MRRR algorithm computes subsets of eigenpairs.) Given a set with k wanted eigenvalues, one can assign non-overlapping subsets of k/p eigenpairs to each processor. This approach is perfect with respect to complexity and memory.

Unfortunately, it does not guarantee numerical accuracy. If the sequential MRRR code (LAPACK's DSTEGR) is called on a given subset, 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. [2]

[2] ScaLAPACK's driver PDSYEVX calls bisection and inverse iteration on non-overlapping subsets

The orthogonality problem can be illustrated nicely by the corresponding representation trees. Figure 3.1 shows the representation tree of a sample matrix when all eigenpairs are computed on a single processor.

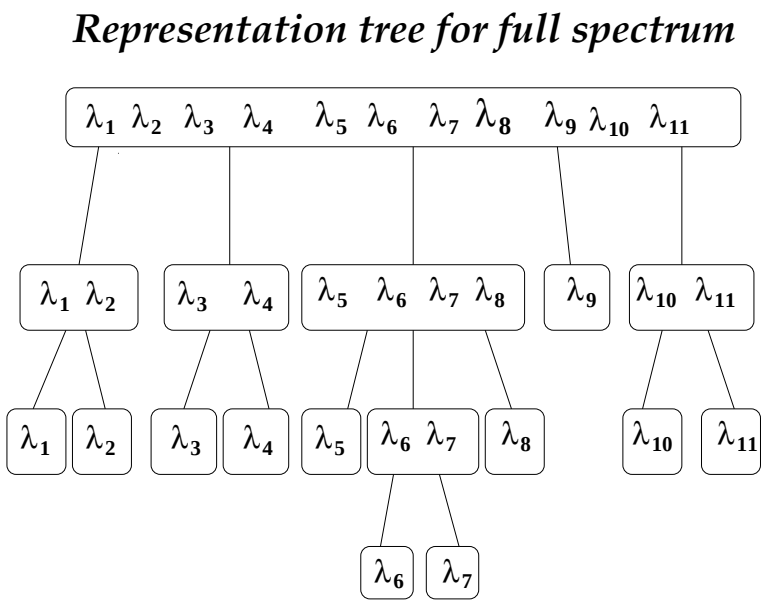

FIG. 3.1. *The (sequential) representation tree for the full spectrum. Square boxes correspond to singletons, rectangular ones correspond to eigenvalue groups for which an individual representation is needed to improve relative gaps.*

Now assume that the computation is parallelized among four processors using the subset approach. The corresponding subset representation trees for each of the processors are shown in Figure 3.2.

It is important to understand the algorithmic significance of the representation tree for the MRRR algorithm. The algorithm only guarantees orthogonality between the eigenvectors computed from the same root representation. Thus, if different root representations are used for different processors, only the eigenvectors within each processor are guaranteed to be orthogonal. Orthogonality across processors depends on the cluster structure and is *not* guaranteed. 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' root representations on all processors. This approach will work, according to the classical gap theorem (3.3), 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 almost all (or all) of the eigenvalues to some accuracy. Thus, in none of its forms, does the subset based approach guarantee

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. One example is the matrix in Appendix B.1.

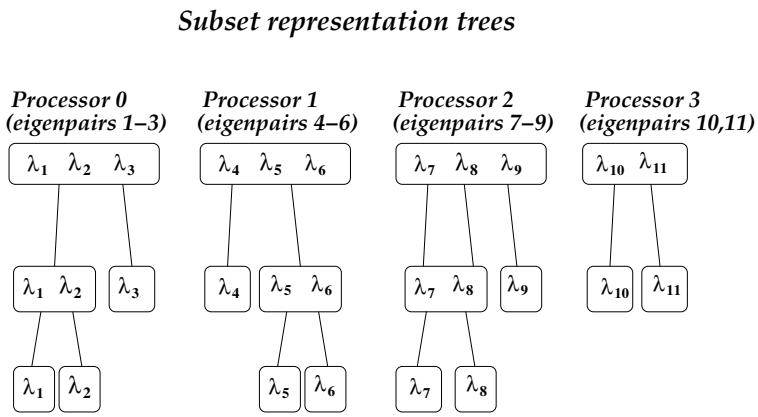

FIG. 3.2. *The subset representation trees by processor for the matrix from Figure 3.1.*

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 [3] 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 of with the same root. Consequentially the parallel algorithm produces 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).

For the ScaLAPACK implementation working in fixed memory, one question remains to be addressed, namely how to store intermediate representations. The MRRR algorithm conventionally stores RRRs for clusters in the eigenvector matrix Z and later overwrites them by the eigenvectors, see [13]. We recently found that it is possible to store the RRRs in the embedded case with (at worst) only one additional vector on each side of the wanted spectrum. This is illustrated in Figure 3.4.

We remark that no additional space is needed if the extremal eigenvalues are close to their neighbors inside the wanted part of the spectrum. Additional storage is only needed if an extremal wanted eigenvalue belongs to another cluster and is separated from the rest of 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 eigenvector storage.

One issue remains. The algorithm requires a superset of the wanted eigenvalues that is well isolated so that the embedding is conformal to the tree for all wanted eigenvalues. Depending on the matrix, the superset can be substantially larger than

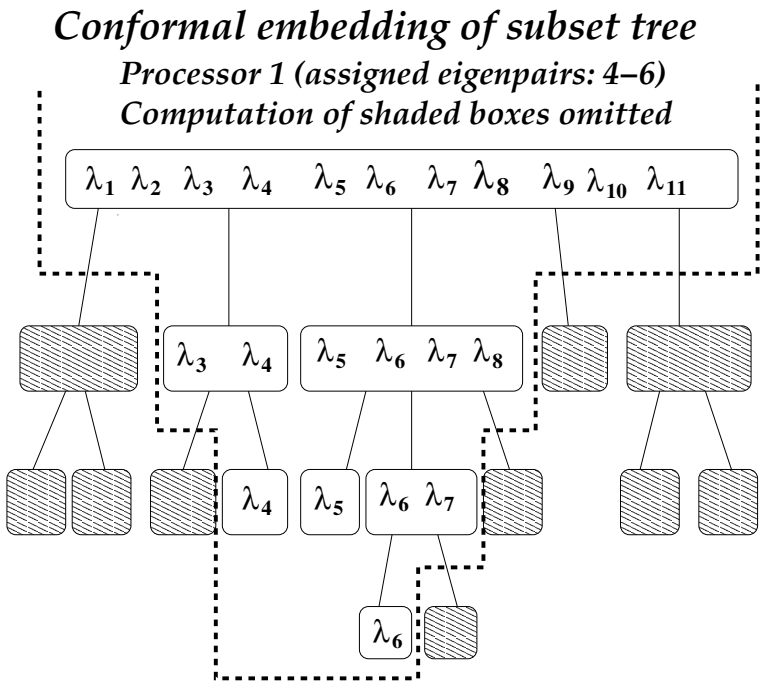

FIG. 3.3. *Illustration of the conformal embedding of the subset representation tree into the representation tree for the full spectrum. (Compare to Figures 3.1 and 3.2.)*

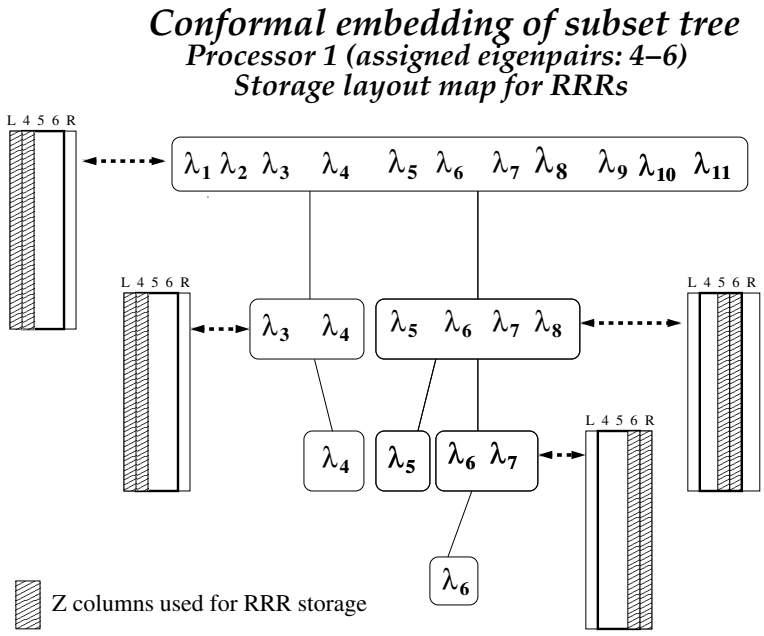

FIG. 3.4. *Storage map for the RRRs in the eigenvector matrix when the subset tree is embedded.*

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.)

3.2.3. Parallelization of the eigenvalue computation. 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.
- Broadcast the eigenvalues to all processors that need them.

It has to be ensured that all processors use the *same* root representations when the eigenvalue computation is parallelized so that the eigenvalues are consistent with the outcome from a sequential computation.

Secondly, it is important to consider what data must be communicated and what data should better be computed redundantly. As a principle, we do not communicate $\mathcal{O}(n)$ data that can be computed by $\mathcal{O}(n)$ work. This principle imposes that each processor should redundantly compute the root representation for the part of the spectrum it is going to be responsible for. Only the eigenvalue computation itself is shared.

It is important to emphasize that it is often faster to compute all eigenvalues redundantly on each processor by a fast sequential algorithm like dqds [27] instead of sharing the work between the processors by parallel bisection, say. See also the remarks in [3]. The reason is that in order to make the parallel eigenvalue computation for the root representation worthwhile, the cost for the subsequent broadcast of the eigenvalues to all processors needs to be amortized.

Thus, the parallel feature for the root eigenvalue computation is optional in our code. Depending on the problem size, the architecture, and the available libraries, the feature can be enabled or disabled. This choice should be done automatically, this is future work.

3.2.4. Requirements of the computing environment. Heterogeneous computing poses enormous difficulties for some of the ScaLAPACK algorithms, see for example [4, 11].

In [4], 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.

In [11], 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.

4. Conclusion and future work. In this paper, we have described our design of PDSYEVR, a ScaLAPACK implementation of the MRRR algorithm to compute the eigenpairs of a symmetric matrix in parallel. Our code guarantees perfect memory scalability ($\mathcal{O}(n^2/p)$ memory per processor) and accuracy for the symmetric eigenvalue problem, in contrast to PDSYEVX in the current ScaLAPACK release. The software is available from the authors on request and will be part of the next ScaLAPACK release.

We now mention three ideas for future work. Firstly, it is necessary to test our code on matrices of larger size than are available in our current tester (described in Appendix A). Secondly, in order to guarantee portable performance, the code should become automatically tunable. Lastly, it would be interesting to compare our code with PDSYEVX and ParEig.

Acknowledgments. We are very grateful to P. Bientinesi for many discussions and comments on the contents of this paper. We also thank J. Demmel, J. Langou, and B. Parlett for their comments on an earlier draft.

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

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

Furthermore, the computed eigenvectors must be numerically orthogonal, satisfying

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

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 [6].

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.

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).

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

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.

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

Appendix C. The interface of pdsyevr.

This section shows the interface of the new ScaLAPACK driver in DOUBLE PRECISION and explains the meaning of each argument.

```
      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 )
*
*     .. Scalar Arguments ..
      CHARACTER          JOBZ, RANGE, UPLO
      INTEGER            IA, IL, INFO, IU, IZ, JA, JZ, LIWORK, LWORK, M,
     $                   N, NZ
      DOUBLE PRECISION   VL, VU
*     ..
*     .. Array Arguments ..
      INTEGER            DESCA( * ), DESCZ( * ), IWORK( * )
      DOUBLE PRECISION   A( * ), W( * ), WORK( * ), Z( * )
*     ..
*
*  Purpose
*  =======
*
*  PDSYEVR computes selected eigenvalues and, optionally, eigenvectors
*  of a real symmetric matrix A by calling the recommended sequence
```

```
*  of ScaLAPACK routines.  Eigenvalues/vectors can be selected by
*  specifying a range of values or a range of indices for the desired
*  eigenvalues.
*
*
*  Arguments
*  =========
*
*  JOBZ    (global input) CHARACTER*1
*          Specifies whether or not to compute the eigenvectors:
*          = 'N':  Compute eigenvalues only.
*          = 'V':  Compute eigenvalues and eigenvectors.
*
*  RANGE   (global input) CHARACTER*1
*          = 'A': all eigenvalues will be found.
*          = 'V': all eigenvalues in the interval [VL,VU] will be found.
*          = 'I': the IL-th through IU-th eigenvalues will be found.
*
*  UPLO    (global input) CHARACTER*1
*          Specifies whether the upper or lower triangular part of the
*          symmetric matrix A is stored:
*          = 'U':  Upper triangular
*          = 'L':  Lower triangular
*
*  N       (global input) INTEGER
*          The number of rows and columns of the matrix A.  N >= 0.
*
*  A       (local input/workspace) block cyclic DOUBLE PRECISION array,
*          global dimension (N, N),
*          local dimension ( LLD_A, LOCc(JA+N-1) )
*
*          On entry, the symmetric matrix A.  If UPLO = 'U', only the
*          upper triangular part of A is used to define the elements of
*          the symmetric matrix.  If UPLO = 'L', only the lower
*          triangular part of A is used to define the elements of the
*          symmetric matrix.
*
*          On exit, the lower triangle (if UPLO='L') or the upper
*          triangle (if UPLO='U') of A, including the diagonal, is
*          destroyed.
*
*  IA      (global input) INTEGER
*          A's global row index, which points to the beginning of the
*          submatrix which is to be operated on.
*
*  JA      (global input) INTEGER
*          A's global column index, which points to the beginning of
*          the submatrix which is to be operated on.
*
```

```
*  DESCA   (global and local input) INTEGER array of dimension DLEN_.
*          The array descriptor for the distributed matrix A.
*          If DESCA( CTXT_ ) is incorrect, PDSYEVR cannot guarantee
*          correct error reporting.
*
*  VL      (global input) DOUBLE PRECISION
*          If RANGE='V', the lower bound of the interval to be searched
*          for eigenvalues.  Not referenced if RANGE = 'A' or 'I'.
*
*  VU      (global input) DOUBLE PRECISION
*          If RANGE='V', the upper bound of the interval to be searched
*          for eigenvalues.  Not referenced if RANGE = 'A' or 'I'.
*
*  IL      (global input) INTEGER
*          If RANGE='I', the index (from smallest to largest) of the
*          smallest eigenvalue to be returned.  IL >= 1.
*          Not referenced if RANGE = 'A'.
*
*  IU      (global input) INTEGER
*          If RANGE='I', the index (from smallest to largest) of the
*          largest eigenvalue to be returned.  min(IL,N) <= IU <= N.
*          Not referenced if RANGE = 'A'.
*
*  M       (global output) INTEGER
*          Total number of eigenvalues found.  0 <= M <= N.
*
*  NZ      (global output) INTEGER
*          Total number of eigenvectors computed.  0 <= NZ <= M.
*          The number of columns of Z that are filled.
*          If JOBZ .NE. 'V', NZ is not referenced.
*          If JOBZ .EQ. 'V', NZ = M
*
*  W       (global output) DOUBLE PRECISION array, dimension (N)
*          On normal exit, the first M entries contain the selected
*          eigenvalues in ascending order.
*
*  Z       (local output) DOUBLE PRECISION array,
*          global dimension (N, N),
*          local dimension ( LLD_Z, LOCc(JZ+N-1) )
*          If JOBZ = 'V', then on normal exit the first M columns of Z
*          contain the orthonormal eigenvectors of the matrix
*          corresponding to the selected eigenvalues.
*          If JOBZ = 'N', then Z is not referenced.
*
*  IZ      (global input) INTEGER
*          Z's global row index, which points to the beginning of the
*          submatrix which is to be operated on.
*
*  JZ      (global input) INTEGER
```

```
*          Z's global column index, which points to the beginning of
*          the submatrix which is to be operated on.
*
*  DESCZ   (global and local input) INTEGER array of dimension DLEN_.
*          The array descriptor for the distributed matrix Z.
*          DESCZ( CTXT_ ) must equal DESCA( CTXT_ )
*
*  WORK    (local workspace/output) DOUBLE PRECISION array,
*          dimension (LWORK)
*          On return, WORK(1) contains the optimal amount of
*          workspace required for efficient execution.
*          if JOBZ='N' WORK(1) = optimal amount of workspace
*             required to compute the eigenvalues.
*          if JOBZ='V' WORK(1) = optimal amount of workspace
*             required to compute eigenvalues and eigenvectors.
*
*  LWORK   (local input) INTEGER
*          Size of WORK
*          See below for definitions of variables used to define LWORK.
*          If no eigenvectors are requested (JOBZ = 'N') then
*             LWORK >= 5 * N + MAX( 12 * NN, NB * ( NP0 + 1 ) )
*          If eigenvectors are requested (JOBZ = 'V' ) then
*             the amount of workspace required is:
*             LWORK >= 5*N + MAX( 18*NN, NP0 * MQ0 + 2 * NB * NB ) +
*               (2 + ICEIL( NEIG, NPROW*NPCOL))*NN
*
*          Variable definitions:
*             NEIG = number of eigenvectors requested
*             NB = DESCA( MB_ ) = DESCA( NB_ ) =
*                  DESCZ( MB_ ) = DESCZ( NB_ )
*             NN = MAX( N, NB, 2 )
*             DESCA( RSRC_ ) = DESCA( NB_ ) = DESCZ( RSRC_ ) =
*                              DESCZ( CSRC_ ) = 0
*             NP0 = NUMROC( NN, NB, 0, 0, NPROW )
*             MQ0 = NUMROC( MAX( NEIG, NB, 2 ), NB, 0, 0, NPCOL )
*             ICEIL( X, Y ) is a ScaLAPACK function returning
*             ceiling(X/Y)
*
*          If LWORK = -1, then LWORK is global input and a workspace
*          query is assumed; the routine only calculates the size
*          required for optimal performance for all work arrays. Each of
*          these values is returned in the first entry of the
*          corresponding work arrays, and no error message is issued by
*          PXERBLA.
*
*  IWORK   (local workspace) INTEGER array
*          On return, IWORK(1) contains the amount of integer workspace
*          required.
*
```

```
*  LIWORK  (local input) INTEGER
*          size of IWORK
*          LIWORK >= 6 * NNP
*          Where:
*            NNP = MAX( N, NPROW*NPCOL + 1, 4 )
*          If LIWORK = -1, then LIWORK is global input and a workspace
*          query is assumed; the routine only calculates the minimum
*          and optimal size for all work arrays. Each of these
*          values is returned in the first entry of the corresponding
*          work array, and no error message is issued by PXERBLA.
*
*  INFO    (global output) INTEGER
*          = 0:  successful exit
*          < 0:  If the i-th argument is an array and the j-entry had
*                an illegal value, then INFO = -(i*100+j), if the i-th
*                argument is a scalar and had an illegal value, then
*                INFO = -i.
*
*  Notes
*  =====
*
*  Each global data object is described by an associated description
*  vector.  This vector stores the information required to establish
*  the mapping between an object element and its corresponding process
*  and memory location.
*
*  Let A be a generic term for any 2D block cyclicly distributed array.
*  Such a global array has an associated description vector DESCA.
*  In the following comments, the character _ should be read as
*  "of the global array".
*
*  NOTATION        STORED IN      EXPLANATION
*  --------------- -------------- --------------------------------------
*  DTYPE_A(global) DESCA( DTYPE_ )The descriptor type.  In this case,
*                                 DTYPE_A = 1.
*  CTXT_A (global) DESCA( CTXT_ ) The BLACS context handle, indicating
*                                 the BLACS process grid A is distribu-
*                                 ted over. The context itself is glo-
*                                 bal, but the handle (the integer
*                                 value) may vary.
*  M_A    (global) DESCA( M_ )    The number of rows in the global
*                                 array A.
*  N_A    (global) DESCA( N_ )    The number of columns in the global
*                                 array A.
*  MB_A   (global) DESCA( MB_ )   The blocking factor used to distribute
*                                 the rows of the array.
*  NB_A   (global) DESCA( NB_ )   The blocking factor used to distribute
*                                 the columns of the array.
*  RSRC_A (global) DESCA( RSRC_ ) The process row over which the first
```

```
*                                 row of the array A is distributed.
*  CSRC_A (global) DESCA( CSRC_ ) The process column over which the
*                                 first column of the array A is
*                                 distributed.
*  LLD_A  (local)  DESCA( LLD_ )  The leading dimension of the local
*                                 array.  LLD_A >= MAX(1,LOCr(M_A)).
*
*  Let K be the number of rows or columns of a distributed matrix,
*  and assume that its process grid has dimension p x q.
*  LOCr( K ) denotes the number of elements of K that a process
*  would receive if K were distributed over the p processes of its
*  process column.
*  Similarly, LOCc( K ) denotes the number of elements of K that a
*  process would receive if K were distributed over the q processes of
*  its process row.
*  The values of LOCr() and LOCc() may be determined via a call to the
*  ScaLAPACK tool function, NUMROC:
*          LOCr( M ) = NUMROC( M, MB_A, MYROW, RSRC_A, NPROW ),
*          LOCc( N ) = NUMROC( N, NB_A, MYCOL, CSRC_A, NPCOL ).
*  An upper bound for these quantities may be computed by:
*          LOCr( M ) <= ceil( ceil(M/MB_A)/NPROW )*MB_A
*          LOCc( N ) <= ceil( ceil(N/NB_A)/NPCOL )*NB_A
*
*  PDSYEVR assumes IEEE 754 standard compliant arithmetic.
*
```

REFERENCES

[1] 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.

[2] 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.

[3] P. Bientinesi, I. S. Dhillon, and R. A. van de Geijn. A Parallel Eigensolver for Dense Symmetric Matrices based on Multiple Relatively Robust Representations. Technical Report TR-03-26, University of Texas, Austin, 2003. To appear in SIAM Journal on Scientific Computing.

[4] 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.

[5] L. S. Blackford, J. 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.

[6] 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).

[7] 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).

[8] 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).

[9] C. Davis and W. Kahan. The rotation of eigenvectors by a perturbation. III. *SIAM J. Numer. Anal.*, 7(1):1–47, 1970.

[10] J. Demmel and J. J. Dongarra. LAPACK working note 164: LAPACK 2005 Prospectus: Reliable and Scalable Software for Linear Algebra Computations on High End Computers. Technical Report UT-CS-05-547, University of Tennessee, Knoxville, 2005.

[11] 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).

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

[13] I. S. Dhillon, B. N. Parlett, and C. Vömel. LAPACK working note 162: The design and implementation of the MRRR algorithm. Technical Report UCBCSD-04-1346, University of California, Berkeley, 2004.

[14] 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).

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] I. C. F. Ipsen. Computing an eigenvector with inverse iteration. *SIAM Review*, 39(2):254–291, 1997.

[20] 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.

[21] B. N. Parlett. *The symmetric eigenvalue problem*. Prentice Hall, Englewood Cliffs, NJ, 1980.

[22] B. N. Parlett. *Acta Numerica*, chapter The new qd algorithms, pages 459–491. Cambridge University Press, 1995.

[23] 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.

[24] B. N. Parlett and I. S. Dhillon. Relatively robust representations of symmetric tridiagonals. *Linear Algebra and Appl.*, 309(1-3):121–151, 2000.

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

[26] B. N. Parlett and I. S. Dhillon. Orthogonal eigenvectors and relative gaps. *SIAM J. Matrix Anal. Appl.*, 25(3):858–899, 2004.

[27] B. N. Parlett and O. Marques. An implementation of the dqds algorithm (positive case). *Linear Algebra and Appl.*, 309:217–259, 2000.

[28] 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. (in preparation).

[29] 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 Verlag, 2000. (also as LAPACK Working Note #139).

[30] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. J. Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1996.

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