SIAM J. Sci. Comput.
Vol. 34, No. 6, pp. C249–C274

TOWARD A HIGH PERFORMANCE TILE DIVIDE AND CONQUER ALGORITHM FOR THE DENSE SYMMETRIC EIGENVALUE PROBLEM*

AZZAM HAIDAR†, HATEM LTAIEF‡, AND JACK DONGARRA§

Abstract. Classical solvers for the dense symmetric eigenvalue problem suffer from the first step, which involves a reduction to tridiagonal form that is dominated by the cost of accessing memory during the panel factorization. The solution is to reduce the matrix to a banded form, which then requires the eigenvalues of the banded matrix to be computed. The standard divide and conquer algorithm can be modified for this purpose. The paper combines this insight with tile algorithms that can be scheduled via a dynamic runtime system to multicore architectures. A detailed analysis of performance and accuracy is included. Performance improvements of 14-fold and 4-fold speedups are reported relative to LAPACK and Intel's Math Kernel Library.

Key words. divide and conquer, symmetric eigenvalue solver, tile algorithms, dynamic scheduling

AMS subject classifications. 15A18, 65F15, 65F18, 65Y05, 65Y20, 68W10

DOI. 10.1137/110823699

1. Introduction.

The objective of this paper is to introduce a new high performance *tile divide and conquer* (TD&C) eigenvalue solver for dense symmetric matrices on homogeneous multicore architectures. The necessity of calculating eigenvalues emerges from various computational science disciplines, e.g., in quantum physics [33], chemistry [37], and mechanics [25], as well as in statistics when computing the principal component analysis of the symmetric covariance matrix. As multicore systems continue to gain ground in the high performance computing community, linear algebra algorithms have to be redesigned or new algorithms have to be developed in order to take advantage of the architectural features brought by these processing units.

In particular, *tile algorithms* have recently shown very promising performance results for solving linear systems of equations on multicore architectures using Cholesky, QR/LQ, and LU factorizations available in the PLASMA [34] library and other similar projects like FLAME [44]. The PLASMA concepts consist of splitting the input matrix into square tiles and reorganizing the data within each tile to be contiguous in memory (block data layout) for efficient cache reuse. The whole dataflow execution can then be represented as a directed acyclic graph (DAG) where nodes are tasks operating on tiles, and edges represent dependencies between them. An efficient and lightweight runtime system environment named QUARK [30] (internally

*Submitted to the journal's Software and High-Performance Computing section February 7, 2011; accepted for publication (in revised form) September 25, 2012; published electronically December 18, 2012. The U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes. Copyright is owned by SIAM to the extent not limited by these rights. This work was partially supported by the National Science Foundation, Department of Energy, and Microsoft Research.

http://www.siam.org/journals/sisc/34-6/82369.html

†Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN (haidar@eecs.utk.edu).

‡KAUST Supercomputing Laboratory, Thuwal, Saudi Arabia (Hatem.Ltaief@kaust.edu.sa).

§Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, and School of Mathematics and School of Computer Science, University of Manchester, Manchester, UK (dongarra@eecs.utk.edu).

used by the PLASMA library) is exploited to dynamically schedule the different tasks and to ensure that the data dependencies are not violated. As soon as the dependencies are satisfied, QUARK initiates and executes the corresponding tasks on the available computational resources. This engenders an out-of-order execution of tasks which removes unnecessary synchronization points and allows different computational stages to overlap. The authors propose to extend these ideas to the symmetric TD&C eigenvalue solver case using a two-stage approach, in which the symmetric dense matrix is first reduced to band form followed by the band divide and conquer (BD&C) eigenvalue solver directly applied on the symmetric band structure.

The resulting TD&C symmetric eigenvalue solver algorithm has been extensively evaluated across many matrix types and against similar D&C symmetric eigenvalue solver implementations from state-of-the-art numerical libraries. The performance results show that the proposed TD&C symmetric eigenvalue solver achieves up to a 14-fold speedup compared to the reference LAPACK [2] implementation and up to a 4-fold speedup compared to the vendor Intel's Math Kernel Library (MKL) [27]. Performance results are also reported comparing the TD&C symmetric eigenvalue solver with other standard methods such as the bisection algorithm, the QR iteration, and the multiple relatively robust representation (MRRR). A study on the accuracy of the computed eigenvalues is provided, which gives a certain confidence on the quality of the overall TD&C symmetric eigenvalue solver framework presented in this paper.

The remainder of this paper is organized as follows: Section 2 recalls some background on symmetric dense matrix eigenvalue solvers and gives a detailed overview of previous projects. Section 3 provides detailed information on the symmetric D&C eigenvalue solver framework applied on band matrices. Section 4 extends the main concepts behind the standard symmetric BD&C eigenvalue solver to instead use tile algorithms associated with a dynamic scheduler. Section 5 presents the algorithmic complexity of the resulting TD&C symmetric eigenvalue solver. Section 6 presents performance results of the overall algorithm on shared-memory multicore architectures against the corresponding standard D&C routines from LAPACK [2] and Intel's MKL V10.2 [27] on various matrix types. Also, performance comparisons against other numerical methods are illustrated, along with a study on the accuracy of the computed eigenvalues. Finally, section 7 summarizes the results of this paper and discusses ongoing work.

2. Background and related work.

This section recalls the standard algorithm to compute the eigenvalues of a symmetric dense matrix and its inherent bottlenecks. One of the eigenvalue solvers is based on the D&C strategy applied to the final condensed tridiagonal form. This section also shows how to extend the D&C eigenvalue solver to compute the eigenvalues from a symmetric band structured matrix, one of the main contributions of the paper.

2.1. The standard symmetric eigenvalue solver approach.

The common way of stating the eigenvalue problem for a symmetric dense matrix is

$$Ax = \lambda x,\ A \in \mathbb{R}^{n\times n},\ x \in \mathbb{R}^n,\ \lambda \in \mathbb{R},$$

with A being a symmetric matrix ($A = A^T$) (Hermitian with $A = A^H$ if $A \in \mathbb{C}^{n\times n}$), λ an eigenvalue, and x the corresponding eigenvector. The goal is first to transform the matrix A into a symmetric tridiagonal matrix T via orthogonal transformations (e.g., Householder reflectors):

$$T = Q \times A \times Q^T, \qquad A, Q, T \in \mathbb{R}^{n\times n}.$$

Once the tridiagonal form is reached, eigenvalues can then be computed using various methods [24, 43], e.g., the bisection algorithm, the QR method, divide and conquer, and MRRR. However, the tridiagonalization step (which requires $\mathcal{O}(n^3)$) is the most time-consuming phase in a symmetric eigenvalue solver. It can reach 90% of the global elapsed time when only the eigenvalues are requested (because the calculation of the eigenvalues requires only $\mathcal{O}(n^2)$) and roughly 50% of the global time when both eigenvalues and eigenvectors are requested (the calculation of the eigenvectors requires $\mathcal{O}(n^3)$).

2.2. Bottlenecks of the standard approach. Because it is the most time-consuming step, it is very important to identify the bottlenecks of the tridiagonalization step, as implemented in LAPACK [2]. The block LAPACK algorithms are characterized by two successive computational steps: the panel factorization and the update of the trailing submatrix. The panel factorization computes the transformations within a specific rectangular region using level 2 BLAS operations (memory-bound) and accumulates them so that they can be applied into the trailing submatrix using level 3 BLAS operations (compute-bound). The parallelism in LAPACK resides within the BLAS library, which follows the expensive fork and join model. This produces artifactual synchronization points between the panel factorization and the trailing submatrix update phases.

In particular, the tridiagonal reduction implemented in LAPACK performs successive panel-update sequences. The symmetric matrix is reduced following a one-stage approach where the reduced form is obtained without intermediate steps. The panel factorization requires loading the entire trailing submatrix into memory. As fast memory is a very scarce resource, this will obviously not scale for large matrices, and thus will generate a tremendous amount of cache and TLB misses. Later, the SBR toolkit [7] introduced a two-stage approach where the matrix is first reduced to a band form during a compute-intensive phase and eventually to the final required form through a memory-bound phase (the bulge-chasing procedure). The two-stage approach permits the casting of expensive memory operations occurring during the panel factorization into faster compute-intensive ones. However, the toolbox still implements block algorithms as in LAPACK and therefore relies on optimized multithreaded BLAS for parallel performance (based on the fork and join paradigm).

2.3. The symmetric eigenvalue solver using D&C approach. Introduced by Cuppen [12], the D&C algorithm computes the eigenvalues of the tridiagonal matrix T. Many serial and parallel Cuppen-based eigensolver implementations for shared and distributed memory have been proposed [18, 22, 26, 28, 38, 40, 42]. The D&C approach can then be expressed in three phases: (a) the partition phase, (b) the solution of the *simple* eigenvalue problems, and (c) the merging phase.

The overall method consists of splitting the problem into two subproblems representing a rank-one tear (a rank-one modification). Each of these subproblems may be considered as an independent problem without any data dependencies on the other subproblems of the same tree level. This process is recursively repeated until the bottom level of the tree, where the bottom nodes will have two independent sons of small size considered as two *simple* eigenvalue problems. This process amounts to constructing a binary tree and is referred to as the partition phase (a). Phase (b) can be described by the independent computation of the eigensystem of all those *simple* eigenvalue problems at the bottom of the tree. Phase (c) consists of merging, on each parent node, the two subproblems (left and right sons) which are defined by a rank-one modification of a diagonal matrix, and then proceeding to the next level with a bottom-up fashion. In fact, each merge (a rank-one modification) consists of computing the eigensystem of a matrix of the form

$$\tilde{Q}\tilde{D}\tilde{Q}^T = (D + \sigma uu^T), \tag{2.1}$$

where D is a real $n \times n$ diagonal matrix, σ is a nonzero scalar, and u is a real vector of order n and has Euclidean norm equal to 1. The eigensystem can be computed based on solving what we call the "secular equation," which restates results of [8, 23, 41, 45], where we refer the reader for more details on the proof. More details on the high-level ideas behind the standard D&C algorithm can be found in [12, 18, 22, 26]. The D&C approach is *sequentially* one of the fastest methods currently available if all eigenvalues and eigenvectors are to be computed [13]. It also has attractive parallelization properties as shown in [42]. Finally, it is noteworthy to mention the *deflation* process, which occurs during the computation of the low rank modifications. It consists of avoiding the computation of an eigenpair of a submatrix (matrix of a son in the tree) that is acceptable to be an eigenpair of the larger submatrix (matrix of a father in the tree). Therefore, the greater the amount of deflations, the lesser the number of required operations, which leads to better performance. The amount of deflations depends on the eigenvalue distribution as well as on the structure of the eigenvectors. In practice, most of the application matrices arising from engineering areas provide a reasonable amount of deflations, and so the D&C algorithm runs at less than $\mathcal{O}(n^{2.5})$ instead of $\mathcal{O}(n^3)$.

2.4. Previous work. The standard D&C eigenvalue solver for symmetric matrices can be extended to start from a symmetric band matrix form. Moreover, band matrices naturally arise in many areas such as the electronic simulations in quantum mechanics, vibrating systems approximated by finite elements or splines, and also in the block Lanczos algorithm, where a sequence of increasingly larger band symmetric eigenvalue problems are generated. Besides the overhead of further reducing the symmetric band matrix to the final tridiagonal form, this motivates attempts to compute the eigendecomposition directly from the band form using variants of the standard D&C algorithm.

Arbenz and coauthors [3, 4, 5] investigated a generalized D&C approach for computing the eigenpairs of symmetric band matrices. The authors provide many important theoretical results concerning the eigenanalysis of a low rank modification of a diagonal matrix. Arbenz [3] proposed two methods for computing eigenpairs of a rank-b modification of a diagonal matrix. The first approach consists of computing the rank-b modification as a sequence of rank-one modifications. The second approach lies in compressing the rank-b modification to a small $b \times b$ eigenproblem, solving it, and then reconstructing the solution of the original problem [4, 5]. The first approach requires more floating point operations (flops) than the second one, which has serial complexity in the $\mathcal{O}(n^3)$ term. The author opted for the second approach. Unfortunately, numerical instabilities in the computation of the eigenvectors have been observed [3], and currently no numerically stable implementation of the second approach exists.

Also, Gansterer, Schneid, and Ueberhuber [19] developed a D&C algorithm for band symmetric matrices which computes the eigendecomposition. Their approach is based on the separation of the eigenvalue and the eigenvector computations. The eigenvalues are computed recursively by a sequence of rank-one modifications of the standard D&C technique. Once the eigenvalues are known, the corresponding eigenvectors are computed using modified QR factorizations with restricted column pivoting.

Later, Gansterer, Ward, and Muller [20] proposed another alternative for the generalized D&C algorithm, which computes *approximate* eigenvalues and eigenvectors of symmetric block tridiagonal matrices. Gansterer et al. [21] then introduced a threshold value τ in the eigensolver algorithm so that the user could decide whether to compute only lower-rank approximations of the off-diagonal blocks or to compute full-rank calculations if higher eigenpair accuracy is desired.

Bai and Ward [6] proposed a parallel and distributed version of [20]. Their algorithm calculates all eigenvalues and eigenvectors of a block-tridiagonal matrix to reduced accuracy by approximating the rank deficient off-diagonal blocks of lower magnitudes with rank-one matrices.

The tile band D&C symmetric eigenvalue solver algorithm (BD&C) presented in this paper differs from Arbenz's algorithm [4, 5] in that it uses the first approach (sequence of b rank-one updates) for the solution of the low-rank modifications along with tile algorithms. The first approach, based on a sequence of b rank-one updates, has also been used by Gansterer, schneid, and Ueberhaber [19]. However, our BD&C algorithm differs from that of [19] in the representation of the subdiagonal blocks, which are not rank deficient. Moreover, our proposed algorithm has been developed in close analogy to the work of Gansterer, Ward, and Muller [20], although we do not compute an approximation of the subdiagonal blocks. We consider the subdiagonal blocks as having a full dense matrix structure, similar to that of Gansterer et al. [21]. The entire algorithm can then be expressed using tiles as a basic block, and once the different computational tasks are defined, a dynamic runtime library is employed to efficiently schedule them on the available resources.

3. The *standard* BD&C symmetric eigenvalue solver.

We assume in this section that the symmetric matrix is originally in band form or has been reduced to band structure by a preprocessing step. We then describe the D&C methodology applied on a symmetric band matrix A, which is divided into p parts. For simplicity, we define $p = 2$, but it is easy to generalize for any $p < n$, with n being the matrix size (see section 3.4).

3.1. Partitioning into subproblems.

Similarly to Cuppen's D&C for tridiagonal matrices, the band matrix A can be divided into p parts. If we split A into two parts, A can be written as follows:

$$A = \begin{pmatrix} B_1 & C_1^T \\ C_1 & B_2 \end{pmatrix}, \tag{3.1}$$

where $B_i \in \mathbb{R}^{p_i \times p_i}$, $p_1 + p_2 = n$, and $C_1 \in \mathbb{R}^{b \times b}$ is the upper triangular block of A, with b the bandwidth size. A can be rewritten in the following form:

$$A = \begin{pmatrix} \tilde{B}_1 & 0 \\ 0 & \tilde{B}_2 \end{pmatrix} + R, \quad \text{where} \quad \tilde{B}_i \in \mathbb{R}^{p_i \times p_i}, \quad p_1 + p_2 = n. \tag{3.2}$$

There are several ways to define the matrix R. It can be shown that the complexity to compute the eigendecomposition of A in the assembly phase is proportional to the square of rank(R). It is therefore important to keep rank(R) as low as possible. Arbenz [3] shows that it is possible to have the rank of $R = b$ if one defines

$$R = \begin{pmatrix} O_{p_1-b} & O & O & O \\ O & M & C_1^T & O \\ O & C_1 & C_1 M^{-1} C_1^T & O \\ O & O & O & O_{p_2-b} \end{pmatrix}, \tag{3.3}$$

where $M = \Theta\Delta\Theta^T$, $\Delta = \Delta^T$, and Δ and Θ are any regular matrices. Gansterer, Schneid, and Ueberhuber [19] have chosen $M = I$. For this choice, R is in most cases unbalanced, meaning that $\| C_1 M^{-1} C_1^T \|$ can be much smaller or larger than $\| M \| = 1$. However, if we choose $M = (C_1^T C_1)^{1/2}$, we get a matrix which is balanced in that sense, and, if $X\Sigma Y^T$ is the singular value decomposition (SVD) of C_1, then $M = Y\Sigma Y^T$ and $C_1 M^{-1} C_1^T = X\Sigma X^T$. Both matrices are symmetric and have the same norm and condition number as C_1. Bai and Ward [6] and Gansterer, Ward, and Muller [20] chose to "*approximate*" the off-diagonal tile C_1 by lower-rank matrices using their SVDs. Later, Gansterer et al. [21] calculated the full-rank of the off-diagonal tiles by default and, depending on the desired eigenpair accuracy, lower-rank approximations of the off-diagonal blocks can be computed instead.

Going back to the standard Cuppen algorithm, we plug in the new notation and A can then be rewritten as

$$A = \begin{pmatrix} \tilde{B}_1 & 0 \\ 0 & \tilde{B}_2 \end{pmatrix} + R = \begin{pmatrix} \tilde{B}_1 & 0 \\ 0 & \tilde{B}_2 \end{pmatrix} + Z\Sigma Z^T, \tag{3.4}$$

where $\tilde{B}_1 = B_1 - Y\Sigma Y^T$, $\tilde{B}_2 = B_2 - X\Sigma X^T$, and $Z = \binom{Y}{X}$. The next step is the computation of the eigendecompositions for each updated block $\tilde{B}_i$.

3.2. Solution of the subproblems. The second phase corresponds to the spectral decomposition of each symmetric diagonal tile ($\tilde{B}_i$). As a result, we can write

$$\tilde{B}_i = Q_{0i} D_{0i} Q_{0i}^T. \tag{3.5}$$

The diagonal matrices D_{0i} contain the eigenvalues of $\tilde{B}_i$, and the matrices Q_{0i} are the corresponding eigenvectors. Thus, the original matrix A can be rewritten as follows:

$$\begin{aligned} A &= \begin{pmatrix} \tilde{B}_1 & 0 \\ 0 & \tilde{B}_2 \end{pmatrix} + Z\Sigma Z^T \\ &= \begin{pmatrix} Q_{01} & 0 \\ 0 & Q_{02} \end{pmatrix} \left\{ \begin{pmatrix} D_{01} & 0 \\ 0 & D_{02} \end{pmatrix} + U\Sigma U^T \right\} \begin{pmatrix} Q_{01} & 0 \\ 0 & Q_{02} \end{pmatrix}^T, \end{aligned} \tag{3.6}$$

where $U = Q_0^T Z$ and $Q_0 = \begin{pmatrix} Q_{01} & 0 \\ 0 & Q_{02} \end{pmatrix}$.

The eigenvalues of A are therefore equal to those of $\{\begin{pmatrix} D_{01} & 0 \\ 0 & D_{02} \end{pmatrix} + U\Sigma U^T\}$. This is equivalent to $(D + U\Sigma U^T)y = \lambda y$, where $U \in \mathbb{R}^{n\times b}$ is a matrix of maximal rank $b \le n$. Thus, having computed the local spectral decompositions of each tree leaf $\tilde{B}_i$, the global spectral decomposition of A can then be calculated by merging the two eigensystems of the two leaves as described in the next subsection.

3.3. Amalgamation of the subproblems. The amalgamation phase consists of traversing the tree in a bottom-up fashion, where the results are merged at each node from the left son and the right son calculations. The merging step handles the computation of the spectral decomposition of $(D + U\Sigma U^T)y = \lambda y$. Here, again, it is obvious that all the merging operations, which perform a *rank-b* updating process between two adjacent eigenvalue subproblems, can concurrently run within the corresponding levels.

There are different methods for computing the eigendecomposition of a low-rank modification of a symmetric matrix $(D + U\Sigma U^T)y = \lambda y$. Two main strategies can be distinguished:

• "$b \times 1$" approach, which computes a sequence of b rank-one modifications of a diagonal matrix.

• "$1 \times b$" approach, which computes a rank-b modification of a diagonal matrix. Arbenz [3] followed the "$1 \times b$" approach to compute the eigenpairs of a rank-b modification by transforming the problem into a smaller $b \times b$ eigenproblem. Unfortunately, numerical instabilities in the computation of the eigenvectors have been observed [3]. The orthogonality of the eigenvectors may be lost if close eigenvalues are present. The resulting algorithm can lose two to four decimal places of accuracy compared to the LAPACK library.

As previously suggested by Gansterer, Schneid, and Ueberhuber [19], one can take advantage of the stability of the existing methodology of the standard D&C algorithm to follow the first approach and compute a sequence of b rank-one modifications for calculating the eigenvalues of $(D + U\Sigma U^T)y = \lambda y$.

The problem to solve now is how to merge the right and the left subproblems. As mentioned above, each merging step consists of a sequence of rank-one updates:

$$A = \begin{pmatrix} \tilde{B}_1 & 0 \\ 0 & \tilde{B}_2 \end{pmatrix} + Z\Sigma Z^T = \begin{pmatrix} \tilde{B}_1 & 0 \\ 0 & \tilde{B}_2 \end{pmatrix} + \sigma_1 z_1 z_1^T + \sigma_2 z_2 z_2^T + \cdots + \sigma_b z_b z_b^T. \tag{3.7}$$

Substituting (3.5) into (3.7) yields

$$\begin{aligned} A &= \begin{pmatrix} Q_{01} & 0 \\ 0 & Q_{02} \end{pmatrix} \left\{ \begin{pmatrix} D_{01} & 0 \\ 0 & D_{02} \end{pmatrix} \right\} \begin{pmatrix} Q_{01} & 0 \\ 0 & Q_{02} \end{pmatrix}^T \\ &\quad + \sigma_1 z_1 z_1^T + \sigma_2 z_2 z_2^T + \cdots + \sigma_b z_b z_b^T \\ &= Q_0 \{D_0\} Q_0^T + \sigma_1 z_1 z_1^T + \sigma_2 z_2 z_2^T + \cdots + \sigma_b z_b z_b^T. \end{aligned} \tag{3.8}$$

Then the rank-b modification will be successively computed as a sequence of b rank-one updates.

First rank-one update:

$$\begin{aligned} A &= Q_0 \, \{D_0 + \sigma_1 u_1 u_1^T\} \, Q_0^T \, + \sigma_2 z_2 z_2^T + \cdots + \sigma_b z_b z_b^T \\ &= Q_0 \, \{Q_1 \, D_1 \, Q_1^T\} \, Q_0^T \, + \sigma_2 z_2 z_2^T + \cdots + \sigma_b z_b z_b^T, \end{aligned} \tag{3.9}$$

where $u_1 = Q_0^T z_1$, D_1 is the rank-one updated eigenvalues of D_0, and Q_1 contains the eigenvectors resulting from this rank-one update.

Second rank-one update:

$$\begin{aligned} A &= Q_0 \, Q_1 \, \{D_1 + \sigma_2 u_2 u_2^T\} \, Q_1^T \, Q_0^T \, + \cdots + \sigma_b z_b z_b^T \\ &= Q_0 \, Q_1 \, \{Q_2 \, D_2 \, Q_2^T\} \, Q_1^T \, Q_0^T \, + \cdots + \sigma_b z_b z_b^T, \end{aligned} \tag{3.10}$$

where $u_2 = Q_1^T \, Q_0^T z_2$, D_2 is the rank-one updated eigenvalues of D_1, and Q_2 contains the eigenvectors resulting from this rank-one updates. This process of rank-one updates is successively applied to all the z_i rank-one vectors $1 \leq i \leq b$.

bth rank-one update:

$$\begin{aligned} A &= Q_0 \, Q_1 \cdots Q_{b-2} Q_{b-1} \, \{D\} \, Q_{b-1}^T Q_{b-2}^T \cdots Q_1^T \, Q_0^T \, + \sigma_b z_b z_b^T \\ &= Q_0 \, Q_1 \cdots Q_{b-2} Q_{b-1} \, \{D + \sigma_b u_b u_b^T\} \, Q_{b-1}^T Q_{b-2}^T \cdots Q_1^T \, Q_0^T \\ &= Q_0 \, Q_1 \cdots Q_{b-2} Q_{b-1} Q_b \, \{\Lambda\} \, Q_b^T Q_{b-1}^T Q_{b-2}^T \cdots Q_1^T \, Q_0^T, \end{aligned} \tag{3.11}$$

where $u_b = Q_{b-1}^T\, Q_{b-2}^T\, \cdots\, Q_1^T\, Q_0^T\, z_b$ and $\Lambda = \text{diag}\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ are the eigenvalues of A.

As a result, once the sequence of b rank-one updates has been computed, we obtain $\Lambda = \text{diag}\{\lambda_1, \lambda_2, \ldots, \lambda_n\}$, which corresponds to the actual eigenvalues of the matrix A.

The whole algorithm for $p > 2$ can be easily described in the following section.

3.4. General case. The band matrix A can be divided into p parts:

$$A = \begin{pmatrix} B_1 & C_1^T & & & \\ C_1 & B_2 & C_2^T & & \\ & C_2 & \ddots & \ddots & \\ & & \ddots & B_{p-1} & C_{p-1}^T \\ & & & C_{p-1} & B_p \end{pmatrix} \in \mathbb{R}^{n\times n}. \tag{3.12}$$

A can be rewritten as follows:

$$A = \text{diag}(\tilde{B}_i) + \sum_{i=1}^{p-1} Z_i \Sigma_i Z_i^T, \tag{3.13}$$

where $X_i \Sigma_i Y_i^T$ is defined as the SVD of C_i, $\tilde{B}_1 = B_1 - Y_1\Sigma_1 Y_1^T$, $\tilde{B}_i = B_i - Y_i \Sigma_i Y_i^T - X_{i-1}\Sigma_{i-1}X_{i-1}^T$ for $2 \le i \le p-1$, $\tilde{B}_p = B_p - X_{p-1}\Sigma_{p-1}X_{p-1}^T$, and

$$Z_1 = \begin{pmatrix} Y_1 \\ X_1 \\ 0 \\ 0 \end{pmatrix}, \quad Z_i = \begin{pmatrix} 0 \\ Y_i \\ X_i \\ 0 \end{pmatrix} \quad \text{for} \quad 2 \le i \le p-2, \quad \text{and} \quad Z_{p-1} = \begin{pmatrix} 0 \\ 0 \\ Y_{p-1} \\ X_{p-1} \end{pmatrix}.$$

In this case, the binary tree has a depth $k = \log_2(p)$ if p is a power of two; $k = \lfloor(\log_2(p))\rfloor + 1$ otherwise. The nodes at the bottom of the tree (level $= k$) are the matrices $\tilde{B}_i$. As mentioned above, the tree is then traversed in a bottom-up fashion, starting from level $k-1$, where at each level l, 2^l independent merging problems are computed. The desired eigenvalues of A are finally computed at the top level $l = 0$, i.e., the root of the tree.

4. The *tile* D&C symmetric eigenvalue solver. This section highlights the skeleton of the tile D&C (TD&C) eigenvalue solver for dense symmetric matrices after recalling the main ideas behind tile algorithms. The dense matrix is first reduced to band form with a given bandwidth b. The three stages of the BD&C symmetric eigenvalue solver (see section 3) are then developed using tile algorithms to calculate all eigenvalues, starting from the symmetric band matrix.

4.1. Concepts of tile algorithms. Tile algorithms have received a lot of interest in recent years for solving linear systems of equations [1, 9, 29, 31, 35, 36]. The main idea consists of overcoming, especially, the fork-join bottleneck of block algorithms, described in section 2.2. The original matrix stored in column-major format is split into a two-dimensional grid of tiles, where the elements are now stored contiguously in memory within each tile (tile data layout), as in Figure 4.1. This may demand a complete reshaping of the standard numerical algorithm. The parallelism is then no longer hidden inside the BLAS routine calls, but rather it is brought to the fore. The whole computation can then be represented as a DAG where nodes

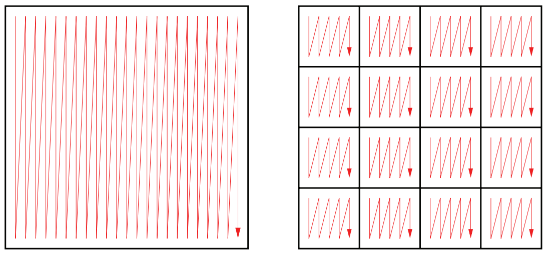


FIG. 4.1. *Translation from LAPACK layout (column-major) to tile data layout.*

represent tasks operating on tiles and edges represent dependencies among them. A dynamic runtime environment system QUARK [30] is employed to ensure the correct processing of all generated tasks at runtime in a holistic fashion and is highlighted in the next section.

4.2. Dynamic scheduling with QUARK. The main, perhaps the most critical, issue when writing a parallel algorithm is how to distribute the data and the work across the available processing units in an efficient way. We have integrated the standalone QUARK [30] framework (engine of the PLASMA [34] library)—a dynamic runtime system environment—into our TD&C symmetric eigenvalue solver algorithm. It is worth mentioning that there exist various scheduling frameworks similar to QUARK, e.g., SuperMatrix [11], part of the libflame [44] library, or SMPSs [39] as a general scheduling library. Our scheduler is designed to use sequential nested-loop code. This is intended to make it easier for algorithm designers to experiment with algorithms and design new algorithms. Each of the original kernel calls (i.e., tasks) is substituted by a call to a wrapper that decorates the arguments with their sizes and their usage (INPUT, OUTPUT, INOUT, VALUE). As an example, in Figure 4.2 we can see how the DGEQRT call (QR factorization) is decorated for the scheduler. The tasks are inserted into the scheduler, which stores them to be executed when all the dependencies are satisfied. That is, a task is ready to be executed when all parent tasks have been completed. The execution of ready tasks is handled by worker threads that simply wait for tasks to become ready and execute them using a combination of default task assignments and work stealing. The thread doing the task insertion is referred to as the master thread. Under certain circumstances, the master thread will also execute computational tasks. Figure 4.3 provides an idealized overview of the architecture of the dynamic scheduler. Moreover, QUARK does not explicitly build the DAG prior to the execution for scalability purposes, but rather unrolls it on the fly within a parametrized window of tasks. Therefore, the execution flow is solely driven by the data dependencies. By providing appropriate hints to QUARK, we are able to resolve two interrelated issues which come into play with regard to scheduling optimality: data locality and the pursuit of the critical path (possibly lookahead opportunities). There are many other details about the internals of the scheduler, including its dependency analysis, memory management, and other performance enhancements that are not covered here. However, information about this scheduler can be found in [30]. The authors would like to use the QUARK's highly productive features, along with applying the concepts of tile algorithms to tackle the challenging two-sided reductions, and in particular, the symmetric *tile* D&C eigenvalue solver. To our knowledge, this is the first time a dynamic runtime system coupled with tile algorithms has been employed to solve one of the two-sided reductions on multicore

```
int QUARK_core_dgeqrt( Quark *quark, int n,
 double *A, int lda, double *T, int ldt, int *info )
{
 QUARK_Insert_Task( quark, TASK_core_dgeqrt, 0x00,
 sizeof(int), &n, VALUE,
 sizeof(double)*n*n, A, INOUT | LOCALITY,
 sizeof(int), &lda, VALUE,
 sizeof(double)*n*n, T, OUTOUT,
 sizeof(int), &ldt, VALUE,
 sizeof(int), info, OUTPUT,
 0);
}
void TASK_core_dgeqrt(Quark *quark)
{
 int n; double *A; int lda; int *T; int ldt; int *info;
 quark_unpack_args_6( quark, n, A, lda, T, ldt, info );
 dgeqrt_( &n, A, &lda, T, &ldt, info );
}
```

Fig. 4.2. *Example of inserting and executing a task in the scheduler. The QUARK_core_dgeqrt routine inserts a task into the scheduler, passing to it the sizes and pointers of arguments and their usage (INPUT, OUTPUT, INOUT, VALUE). Later, when the dependencies are satisfied and the task is ready to execute, the TASK_core_dgeqrt routine unpacks the arguments from the scheduler and calls the actual dgeqrt factorization routine.*

architecture.

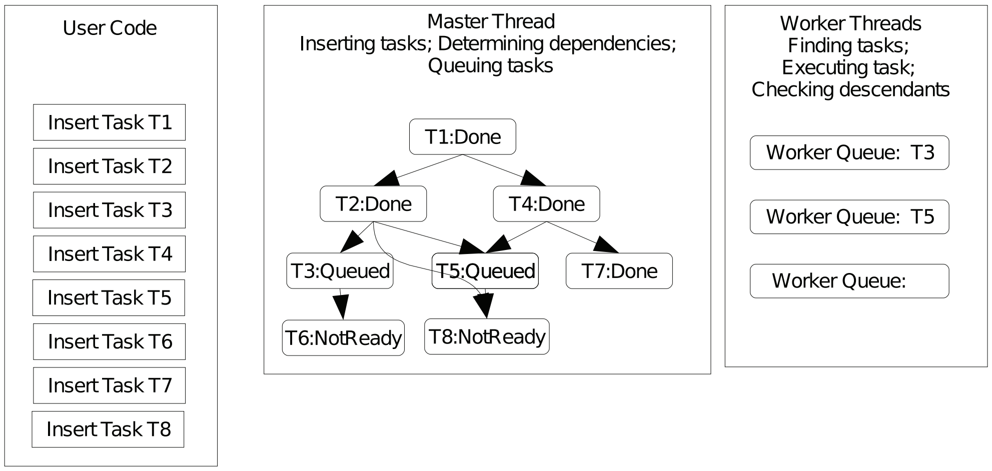


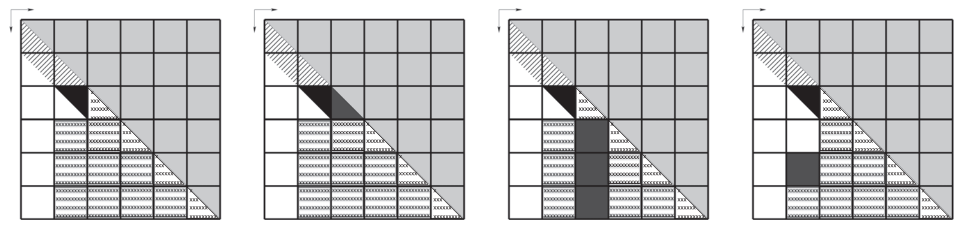
Fig. 4.3. *Idealized architecture diagram for the dynamic scheduler. Inserted tasks go into an (implicit) DAG based on their dependencies. Tasks can be in NotReady, Queued, or Done states. Workers execute queued tasks and then determine if any descendants have now become ready and can be queued.*

4.3. First step: Symmetric matrix reduction to band form.

This section describes the numerical kernels involved in the reduction to symmetric band form and highlights the grouping technique used to further enhance data locality during this first step while achieving a small bandwidth size.

4.3.1. Description of the numerical kernels.

The matrix reduction to symmetric band form follows the tile algorithm strategy developed for the QR algorithm

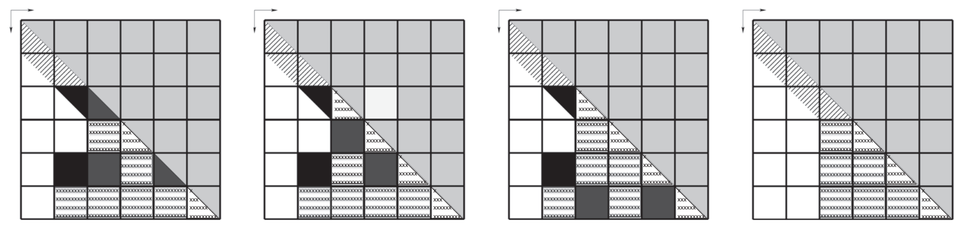


(a) xGEQRT: QR factorization of a single tile.

(b) xSYRFB: Left/right updates on a symmetric diagonal tile.

(c) xORMQR: Right updates of a single tile down to the bottom.

(d) xTSQRT: QR factorization of a triangular tile on top a square tile.



(e) xTSMQRLR: Left-/right applications on the diagonal region.

(f) xTSMQR: Left application using the transpose of the light gray upper tile.

(g) xTSMQR: Right application down to the bottom.

(h) Matrix structure at the end of the reduction step 2.

FIG. 4.4. *Kernel execution breakdown of the TD&C algorithm during the first stage.*

[10] and relies on highly optimized compute-intensive kernels to achieve high performance. It is composed of eight kernels total. Four kernels come directly from the one-sided QR factorization [10], and the four others have been recently implemented to handle the symmetry property of the matrix when updating the trailing submatrix around the diagonal.

Figure 4.4 highlights the execution breakdown at the second step of the reduction to band form. Since the matrix is symmetric, only the lower part is referenced, and the upper part (gray color) stays untouched. We recall that xGEQRT computes a QR factorization of a single subdiagonal tile, as presented in Figure 4.4(a). Then the left and right applications of a Householder reflector block on a symmetric diagonal tile is done by the new xSYRFB kernel, as shown in Figure 4.4(b). The right applications on the single tiles then proceed along the same tile column using the xORMQR kernel, as depicted in Figure 4.4(c). Once this is done, we start the annihilation of the tiles below the subdiagonal, one by one using the xTSQRT kernel, and for each tile annihilated we will apply its corresponding left and right updates. Figure 4.4(d) shows an example of how xTSQRT computes a QR factorization of a matrix composed by the subdiagonal tile (3,2), and the square tile located below it (5,2), on the same tile column. Once the Householder reflectors have been calculated, they need to be applied from left and right to the trailing submatrix.

On the example illustrated in Figure 4.4, we need to apply the Householder reflectors from left to two rows of tiles formed by row1(3,3:5) and row2(5,3:5) taking care of the symmetry, and from right on the two columns of tiles formed by col1(3:6,3) and col2(3:6,5) taking special care of the symmetry. For that, we developed a new kernel to appropriately handle the symmetric property of the trailing matrix. Indeed, the xTSMQRLR kernel in Figure 4.4(e) loads three tiles together—two of them are symmetric diagonal tiles—and carefully applies left and right orthogonal transformations on them.

Once the special treatment for the symmetric diagonal structure has completed, we then apply the same transformations to the remaining left side of the matrix as well as to the right side (down to the bottom of the matrix) using the xTSMQR kernel with the corresponding side variants, as displayed in Figures 4.4(f) and 4.4(g), respectively. The remaining left side is between tiles (3,4) and (5,4). It is necessary to further explain the left variant, as it should require the light gray tile located in position (3,4) from Figure 4.4(f). In fact, this tile is not referenced since only the lower part of the matrix is being operated. Therefore, by taking its transpose located in position (4,3), we can apply the "left" on tile $(4,3)^T$ and tile (5,4) updating (5,4). But, because of the symmetry, this later is similar to the "right" on tile (4,3) and tile (4,5), and thus by doing the "left" on tile $(4,3)^T$ and tile (5,4), updating both of them, we are able to compute the "right" operations on (4,3). As result, only the "left" application is done updating both tiles, and thus the remaining right side becomes tiles (6,3) and (6,5). Finally, Figure 4.4(h) represents the matrix structure at the end of the second step of the reduction. The symmetric band structure starts to appear at the top left corner of the matrix (the dashed area).

Since the entire algorithm operates on tiles, the tile size b naturally corresponds to the bandwidth size. This matching between tile and bandwidth sizes is in fact more than just a choice for productivity purposes. Indeed, it is an algorithmic restriction to avoid destroying the obtained zeroed structure while computing the symmetric band form. For instance, in the lower symmetric case, the left updates have to be shifted one tile down (i.e., the actual bandwidth size) from the diagonal tile such that the right ones do not produce fill-in elements in the already annihilated part of the matrix. Thus, tuning the parameter b is critical, as explained in the next section.

4.3.2. Enhancing data locality by grouping tasks. A small bandwidth b will dramatically affect the efficiency of the reduction phase and increase the elapsed time due to (1) the huge bus traffic required to load all the kernel data into memory, (2) the inefficiency of the kernels on small b, and (3) the runtime system overhead of scheduling very fine granularity tasks. At the same time, as analyzed later in section 5, the number of flops of the BD&C symmetric eigenvalue solver strongly depends on the band size b. The smaller the b, the fewer the number of operations. This trade-off between achieving high performance for the reduction step and diminishing the number of flops of the band D&C symmetric eigenvalue solver must then be addressed.

To overcome those three limitations, we have implemented a *task grouping technique*, which consists of aggregating different data tiles as well as the computational kernels operating on them, in order to build a *super* tile. The idea is that the kernels operate on a super tile overcoming limitations (1) and (2), also increasing data reuse where we are forced to apply all the possible operations that could be applied to the super tile. Moreover, such a technique allows us to decrease the number of tasks, as all the operations occurring on a super tile are considered as only one task. This later helps us to sweep over limitation (3) by removing the scheduler overhead. Figure 4.5 shows the super tiles of size 2. The kernels operating on those super tiles are actually a combination of the kernels used without the grouping technique. For instance, in Figure 4.5, to enhance the data locality, the single call to the update kernel on the dark gray super tiles will be internally decoupled by multiple calls to the corresponding kernel variants (i.e., left update, followed by a left transposed and right updates). For instance, the elapsed time for reducing a symmetric dense matrix

($n > 4000$) to band form with $b = 20$ on an Intel Xeon system using 16 cores can be reduced up to one order of magnitude by enabling the task grouping technique, and the number of tasks can decrease up to two orders of magnitude compared to the original version, i.e., without the task grouping technique. Therefore, applying the task grouping technique permits us to substantially improve the reduction phase and, at the same time, to obtain a small bandwidth size, which is necessary for the second step to be effective.

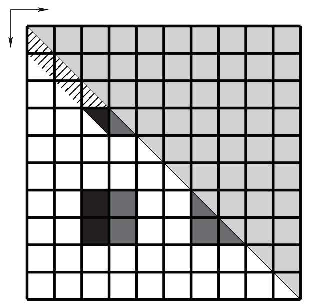


FIG. 4.5. *A schematic description of the task grouping technique using super tiles of size* 2.

Once the band form is obtained, the goal is to compute the eigenvalues directly from the band form using the BD&C symmetric eigenvalue solver, which is the main topic of the next section.

4.4. Second step: Computing all eigenvalues from the band form using BD&C. Applying the D&C algorithm on the symmetric band form using tile algorithms turns out to be straightforward, thanks to the high level of productivity provided by the QUARK framework. The partitioning into subproblems consists of computing the SVDs of each off-diagonal tile, which are independent from each other and can therefore run concurrently. Once the SVDs are computed, the symmetric diagonal tiles are then updated accordingly, and the solutions of each subproblem can be triggered by calculating the spectral decompositions. Here, again, the spectral decompositions are completely independent and can run in parallel. The computational routines used for the two phases are DGESVD and DSYEVD from LAPACK, respectively. The last phase consists of the amalgamation of the subproblems in which the contributions of a pair of subproblems are merged together (b rank-one modifications) following a reverse tree traversal. A customized kernel has been implemented to merge the different subproblems, which are independent from each other when located on the same tree level.

The three phases can thus be easily mapped into the tile algorithms. It is also noteworthy to mention that a dynamic-type scheduler is preferred over a static one, especially since the work of the BD&C algorithm depends strongly on the amount of deflations, which can produce load balancing issues. Therefore, there is no need to wait until the end of the merging procedure of all subproblems within a tree level. The merge of the subsequent subproblems at higher level of the tree can proceed as soon as the previous subproblem amalgamations terminate. Moreover, the fine granularity of tile algorithms, together with the dynamic scheduler, permits the removal of the artifactual synchronization points seen in the standard D&C parallel implementation located (1) inside the fork/join paradigm and (2) between the different phases of the algorithm. Tile algorithms alleviate (1) by bringing the parallelism to the fore and relying on sequential high performance kernels with a smaller granularity. Since the execution flow of the BD&C symmetric eigenvalue solver through QUARK is solely driven by the data dependencies, a task immediately proceeds after the dependencies coming from its parent are satisfied, regardless of the application global status. Therefore, the unnecessary synchronization points between the different phases of the algorithm are completely removed, solving (2). Tasks from the reduction phase and from the BD&C phase can substantially overlap. The BD&C phase can in fact start even before the dense matrix has completely achieved the band form.

5. Algorithmic complexity.

In this section, we calculate the number of flops required by the TD&C symmetric eigenvalue solver algorithm. We recall that b corresponds to the tile and the bandwidth sizes. The matrix is divided into p blocks/tiles, i.e., $p = n/b$, with n being the size of the matrix.

The algorithmic complexity of the full tridiagonal reduction is $\mathcal{O}(4/3n^3)$. The calculation of the number of flops for the band reduction step is then straightforward. It requires $4/3n \times (n-b) \times (n-b)$ flops and therefore gets closer to $\mathcal{O}(4/3n^3)$ for small bandwidth size b. The flop calculation for the BD&C algorithm to get the actual eigenvalues is more complicated. The flop count for each phase is reported below:

- *Partitioning phase.* This phase is characterized by the SVD of the off-diagonal blocks (estimation bound for small square matrices of size b is $6.67b^3$) and the construction of the new diagonal blocks $\tilde{B}_i$ as defined by (3.4). The number of flops is then at most $(p-1) \times (6.67)b^3 + (p-1) \times 2 \times 2b^3$.
- *Subproblems solution phase.* This phase computes the spectral decompositions of each symmetric $\tilde{B}_i$ as described by (3.5). Its cost depends on the algorithm used and the convergence of the eigenvalues (an upper bound for small square matrices of size b is $9b^3$ [24]). It requires at most $p \times 9b^3$ flops.
- *Amalgamation phase.* This phase consists of the merging of the son nodes of a giving level, two by two, starting from level $k-1$ up to level 0. Each merging step of two-by-two nodes involves the application of a sequence of b rank-one modifications. Each rank-one modification necessitates the computation of a rank-one vector $u_i = Q_i^T z_i$ and an eigensolution of $D + \sigma_i u_i u_i^T$. The computation of the rank-one vector u_i involves a set of matrix-vector products with the computed Q_i^T from the current level (cost detailed in α) and also with all the Q_i^T from the previous deeper levels of the tree (cost detailed in β). The cost of the eigensolution will be reported in γ.
  - α – For the current level, for each rank-one modification "j," as described by (3.11), we will have to apply a matrix-vector product with all the previously computed Q_r^T (where $r = 1, \dots, j-1$) of the current level. Thus, for the b rank-one modifications, this gives rise to $\frac{b}{2}(b-1)$ matrix-vector products of a matrix of size $(\frac{n}{2^i})$. Note that for each level there are 2^i leaves to be amalgamated, giving rise to $2^i \times \frac{b}{2}(b-1) \times 2(\frac{n}{2^i})^2$. Thus, for all the levels $(0, \dots, k-1)$, this is expressed by the first component of the formula below.
  - β – The cost of the matrix-vector product with the Q_i^T from the previous deeper levels of the tree can be described as follows:
    For each level, we don't store the Q matrices; thus for each Q generated, we multiply it by all the vectors u of the upper parents. Thus, for a given level i, we generate b matrices Q of size $n/(2^i)$, and each of these matrices is multiplied by all the u vectors of the upper parent in the tree. In other words, each Q is multiplied by the b vectors u of each upper

level of the tree. For a given level i, we will have i upper level, and thus Q is multiplied by ib vectors u. We generate b matrices Q of size $n/(2^i)$, so overall, for a level i, we will have $b \times ib$ matrix-vector products of size $(\frac{n}{2^i})$. The cost is $b \times ib \times 2(\frac{n}{2^i})^2$, and for the 2^i submatrices Q, $2^i \times b \times ib \times 2(\frac{n}{2^i})^2$. Thus for all the levels, where we have to multiply the generated matrices Q by those of upper parents (levels $1, \dots, k-1$), the cost is represented by the second term of the formula below.

γ – The eigendecomposition of $D + \sigma uu^T \in \mathbb{R}^{r \times r}$ demands two distinct computation steps: the zero finding algorithm, which requires an average of $5r^2$ operations, and the calculation of r eigenvectors, which costs $2r^2$ operations. The number of flops involved in the eigensolution of a rank-one modification is $\mathcal{O}(7r^2)$. In practice, this number is the worst-case scenario, as the remarkable phenomenon of deflation takes place and may dramatically reduce this cost depending on the number of nondeflated vectors. Finally, for b rank-one modifications, for all the 2^i leaves and for all the levels, the cost is reported in the third component of the formula below.

Now, if we sum up all eigendecomposition computations occurring at all tree levels with a depth $k = \lfloor \log_2(p) \rfloor$, the cost of the amalgamation step is equal to

$$\sum_{i=0}^{k-1} 2^i \times \frac{b}{2}(b-1) \times 2\left(\frac{n}{2^i}\right)^2 + \sum_{i=1}^{k-1} 2^i \times b \times ib \times 2\left(\frac{n}{2^i}\right)^2 + \sum_{i=0}^{k-1} 2^i \times b \times 7\left(\frac{n}{2^i}\right)^2$$

$$= \sum_{i=0}^{k-1} \frac{b}{2}(b-1) \times 2n^2 \left(\frac{1}{2}\right)^i + \sum_{i=1}^{k-1} b \times ib \times 2n^2 \left(\frac{1}{2}\right)^i + \sum_{i=0}^{k-1} b \times 7n^2 \left(\frac{1}{2}\right)^i$$

$$\approx \frac{b^2}{2} \times 2n^2 \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i + 2b^2n^2 \sum_{i=1}^{k-1} i \left(\frac{1}{2}\right)^i + 7bn^2 \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i$$

We know that $\sum_{i=0}^{\infty} ar^i = \frac{a}{1-r}$ and $\sum_{i=0}^{\infty} ir^i = \frac{r}{(1-r)^2}$ for $|r| < 1$, and thus

$$\approx 2b^2n^2 + 4b^2n^2 + 14bn^2 \approx \mathcal{O}(6\ b^2n^2) \text{ flops.}$$

The total arithmetic complexity of the BD&C algorithm will be dominated by the amalgamation phase and will require at most $\mathcal{O}(6\ b^2n^2)$ flops.

We note that the cost of the TD&C for computing all eigenvalues of a dense symmetric matrix is $\mathcal{O}(4/3n^3 + 6\ b^2n^2)$. It is straightforward to see that the size of the bandwidth b will have a huge impact on the overall algorithm complexity. The TD&C symmetric eigenvalue solver algorithm will be cost-effective if and only if $b \lll n$. Also, if all corresponding eigenvectors are to be calculated, the TD&C symmetric eigenvalue solver will not be suitable, as the cost will be dominated by the accumulation of the intermediate eigenvector matrices for each single rank-one modification. This would involve matrix-matrix multiplications, and thus the overall cost will substantially increase and become dominated by $\frac{8}{3}bn^3$ flops. Although these level 3 BLAS operations are well suited for hardware accelerators such as GPUs, other methods, such as MRRR [16, 17], may be more competitive and need to be explored if the eigenvectors are required.

6. Performance results and accuracy analysis.

This section summarizes our main performance results of the TD&C symmetric eigenvalue solver algorithm.

6.1. Machine description. All of our experiments have been run on a shared-memory multicore architecture composed of a quad-socket quad-core Intel Xeon EMT64 E7340 processor operating at 2.39 GHz. The theoretical peak is equal to 9.6 Gflop/s per core or 153.2 Gflop/s for the whole node, composed of 16 cores. The practical peak achieved with DGEMM on a single core is equal to 8.5 Gflop/s or 136 Gflop/s for the 16 cores. The level-1 cache, local to the core, is divided into 32 kB of instruction cache and 32 kB of data cache. Each quad-core processor is actually composed of two dual-core Core2 architectures, and the level-2 cache has 2×4 MB per socket (each dual-core shares 4 MB). The machine provides Intel Compilers 11.0 together with the Intel MKL V10.2 vendor library.

6.2. Tuning the tile/band size *b*. From sections 4 and 5, it is clear that the tile/band size is the paramount parameter. Figure 6.1 highlights the effect of this parameter on the reduction phase (line with squares), on the BD&C phase (line with pluses), and on the overall TD&C eigenvalue solver (line with circles). For a very large b, the BD&C phase is the dominant part of the general algorithm and, reciprocally, for a very small b, the reduction phase governs the whole application. From these experiments, a tile size $b = 20$ looks to be the best compromise for a matrix of order less than 20000, while $30 < b < 40$ appears to be the best choice for a matrix of larger order.

6.3. Performance comparisons of the first stage reduction from dense to band. Figure 6.2(a) highlights the performance of the first stage (PLASMA-DSYRDB) when varying the semibandwidth size. We compare our first stage procedure against the equivalent function from the SBR toolbox using two different tile sizes, i.e., $b = 200$ and $b = 20$. Our implementation of the first stage performs very well, thanks to the increased parallelism degree brought by the tile algorithms. Although this stage is very computationally intensive, we could easily note that the performance of the first stage is very sensitive to the semibandwidth size. The performance decreases as the tile size gets smaller; however, it remains sustainably acceptable. The equivalent SBR function to reduce the symmetric dense to band form does not

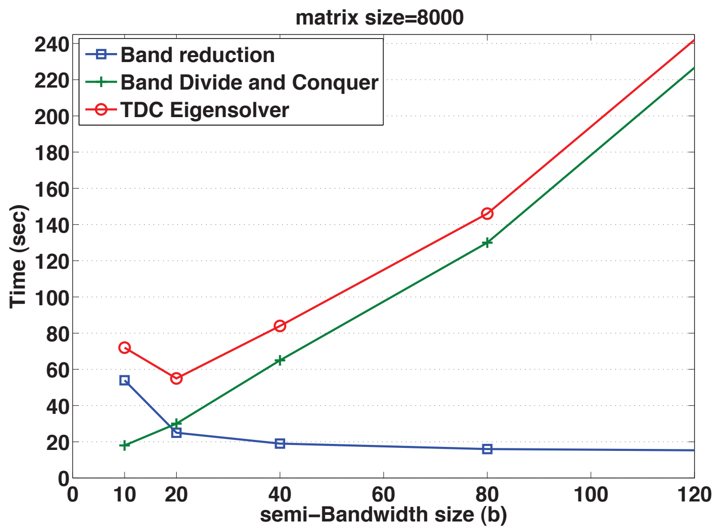
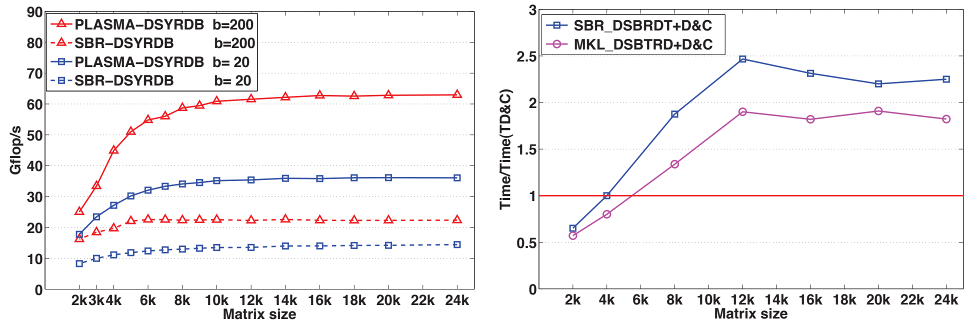


FIG. 6.1. *Effect of the tile/band size.*



(a) Performance comparisons of the first stage when varying the semibandwidth size. (b) Performance comparisons of finding the eigenvalues of band matrices with a bandwidth equal to 20.

FIG. 6.2. *Kernel execution breakdown of the TD&C algorithm during the first stage.*

perform similarly, surprisingly, mainly due to the overhead of the nested parallelism within the fork-join paradigm.

6.4. Performance comparisons of the second stage finding the eigenvalues of band matrices. Figure 6.2(b) presents timing comparisons in seconds of the second stage, finding the eigenvalue of band matrices using either our BD&C or the standard techniques of reducing the band matrices to tridiagonal, then finding the eigenvalue of the tridiagonal matrices. The elapsed time of the standard techniques is divided by the elapsed time of our BD&C. We compare against the MKL-DSBTRD and also against the SBR-DSBRDT, which reduce a band matrix to tridiagonal. The bandwidth size is equal to 20, which corresponds to our choice of optimal bandwidth.

6.5. Performance comparisons with other symmetric D&C implementations. This section shows the performance results of our TD&C symmetric eigenvalue solver and compares them against the similar routine using D&C algorithms available in the state-of-the-art open source and vendor numerical libraries, i.e., multithreaded reference LAPACK V3.2 with optimized Intel MKL BLAS and Intel MKL V10.2, respectively. Since the eigensolver routines used (DSTEDC) operate on a tridiagonal matrix, the dense symmetric matrix first needs to be reduced to tridiagonal form. Intel MKL actually provides two interfaces to reduce the symmetric dense matrix to tridiagonal form. The first corresponds to the optimized version of DSYTRD from LAPACK (MKL-LAPACK), and the second one integrates the optimized version of the corresponding routine named DSYRDD from the SBR toolbox [7] (MKL-SBR). Figure 6.3 summarizes the experiments in Gflop/s. The number of flops used as a *reference is* $4/3n^3$, which is roughly the total number of flops for the tridiagonal reduction, since the D&C eigenvalue solver of a tridiagonal matrix is of order n^2. All experiments have been performed on 16 cores, with random matrices, in double precision arithmetic.

The performance results show that the proposed TD&C symmetric eigenvalue solver achieves up to a 14-fold speedup compared to the reference LAPACK implementation, and up to a 2.5-fold speedup as compared to the vendor Intel MKL library on random matrices.

The next sections describe a collection of different matrix types and compare our TD&C symmetric eigenvalue solver against D&C (DSYTRD+DSTEDC) as well as other methods to compute the eigenvalues of a symmetric dense matrix, e.g., the bisec-

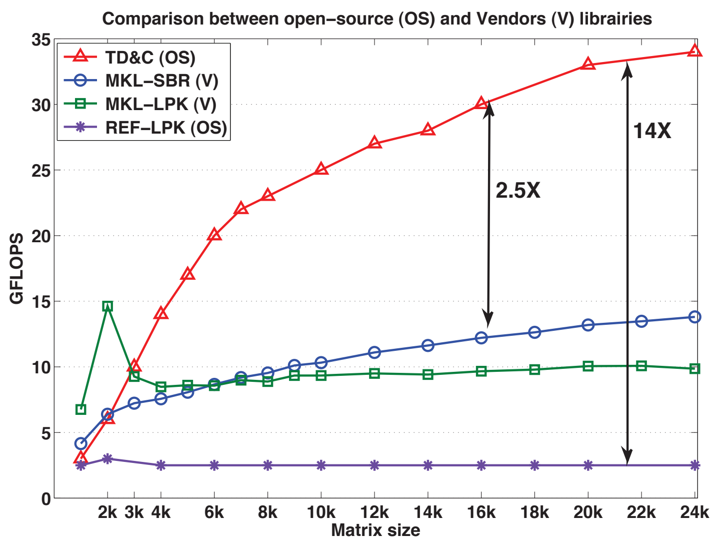


FIG. 6.3. *Performance comparisons of the TD&C symmetric eigenvalue solver against open-source (OS) and vendor (V) libraries.*

tion BI (DSYTRD+DSTEVX), QR (DSYTRD+DSTEV), and MRRR (DSYTRD+DSTEMR). For simplicity, in the experiments shown in the next sections, we compare our implementation with those four routines from the Intel MKL and skip the comparison with the reference LAPACK library.

6.6. Description of the matrix collections. In this section, we present the different matrix collections used during the extensive testing to get performance results (section 6.7) as well as the numerical accuracy analysis (section 6.8). There are three matrix collections:

• The matrices from the first set, represented in Table 6.1, are *synthetic testing matrices* chosen to have extreme distributions of eigenvalues as well as other specific properties that exhibit the strengths and weaknesses of a particular algorithm. More information about these matrix collections can be found in [14, 15, 32].

TABLE 6.1
Synthetic testing matrices from LAPACK *testing (types 1–6) and from* [15] *(types 7–9). Note that for distributions of types 1–5, the parameter k has been chosen to be equal to one over the machine precision $1/ulp$ in double precision arithmetic. We use the* LAPACK *routines* DLATMS *or* DLAGSY *to generate $A = QDQ^T$. Given the eigenvalues λ, the dense matrix A is generated by multiplying $D = \mathrm{diag}(\lambda)$ by an orthogonal matrix Q generated from random entries.*

	Description
Type 1	$\lambda_1 = 1, \quad \lambda_i = \frac{1}{k}, \quad i = 2, 3, \ldots, n$
Type 2	$\lambda_i = 1, \quad i = 2, 3, \ldots, n-1, \quad \lambda_n = \frac{1}{k}$
Type 3	$\lambda_i = k^{-(\frac{i-1}{n-1})}, \quad i = 2, 3, \ldots, n$
Type 4	$\lambda_i = 1 - (\frac{i-1}{n-1})(1 - \frac{1}{k}), \quad i = 2, 3, \ldots, n$
Type 5	n random numbers in the range $(\frac{1}{k}, 1)$; their logarithms are uniformly distributed
Type 6	n random numbers from a specified distribution
Type 7	$\lambda_i = ulp \times i, \quad i = 1, 2, \ldots, n-1, \quad \lambda_n = 1$
Type 8	$\lambda_1 = ulp, \quad \lambda_i = 1 + i \times \sqrt{ulp}, \quad i = 2, 3, \ldots, n-1, \quad \lambda_n = 2$
Type 9	$\lambda_1 = 1, \quad \lambda_i = \lambda_{i-1} + 100 \times ulp, \quad i = 2, 3, \ldots, n$

Table 6.2
Matrices with interesting properties.

	Description
Type 10	(1,2,1) tridiagonal matrix
Type 11	Wilkinson-type tridiagonal matrix
Type 12	Clement-type tridiagonal matrix
Type 13	Legendre-type tridiagonal matrix
Type 14	Laguerre-type tridiagonal matrix
Type 15	Hermite-type tridiagonal matrix

Table 6.3
Matrices from real-life applications.

	Description
Type 16	Matrices from an application on quantum chemistry and electronic structure
Type 17	The bcsstruc1 set in the Harwell–Boeing collection
Type 18	Matrices from the Alemdar, NASA, cannizzo, etc., sets at the University of Florida

• The second set of matrices are *matrices with interesting properties*, which are represented in Table 6.2. Those matrices will only be used for the accuracy analysis in section 6.8 since they are already in tridiagonal form.

• The third set are *practical matrices*, which are based on a variety of practical applications, and thus are relevant to a large group of users. Some of these matrices are described in Table 6.3.

6.7. Performance comparisons with other symmetric eigenvalue solvers. This section compares the TD&C approach with other state-of-the-art symmetric eigenvalue solvers.

6.7.1. Understanding the various graphs. All experiments have been performed on 16 cores with matrix types from Tables 6.1 and 6.3. The matrix sizes vary from 1000 to 24000, and the computation is done in double precision arithmetic. Our TD&C symmetric eigenvalue solver is compared with the bisection BI, QR iteration, and MRRR ("MR"), as well as with the D&C symmetric eigenvalue solvers. Similarly to DSTEDC, the other eigenvalue solver implementations necessitate the matrix being reduced into tridiagonal form using either the MKL LAPACK-implementation routine DSYTRD, or the MKL SBR-implementation routine DSYRDD. The graphs represents the *speedup* obtained by computing the ratio between the elapsed time of these four methods over the TD&C total execution time—in other words, a *value t above* 1 means that our TD&C is *t times faster* than the corresponding algorithm, and a *value t below* 1 means that our TD&C is $1/t$ *times slower* than the corresponding algorithm. The symbol codes used for all plots are as follows: BI is denoted by "$\times$", QR is denoted by "*", MR is denoted by "$\diamond$", D&C is denoted by "$\circ$", and TD&C is denoted by a thick horizontal line considered as a reference for the speedup graphs.

6.7.2. Synthetic testing matrices (Table 6.1). Figure 6.4 shows the speedup obtained by the TD&C symmetric eigenvalue solver as compared to BI, QR, MR, and D&C when the MKL LAPACK-implementation routine DSYTRD is used to reduce the first stage. Figure 6.5 shows the speedup obtained by the TD&C symmetric eigenvalue solver as compared to BI, QR, MR, and D&C when the MKL SBR-implementation routine DSYRDD is used to reduce the first stage. In this case we can expect that the performance obtained will have a similar trend to the ones using the MKL LAPACK DSYTRD presented in Figure 6.4, but slightly better due

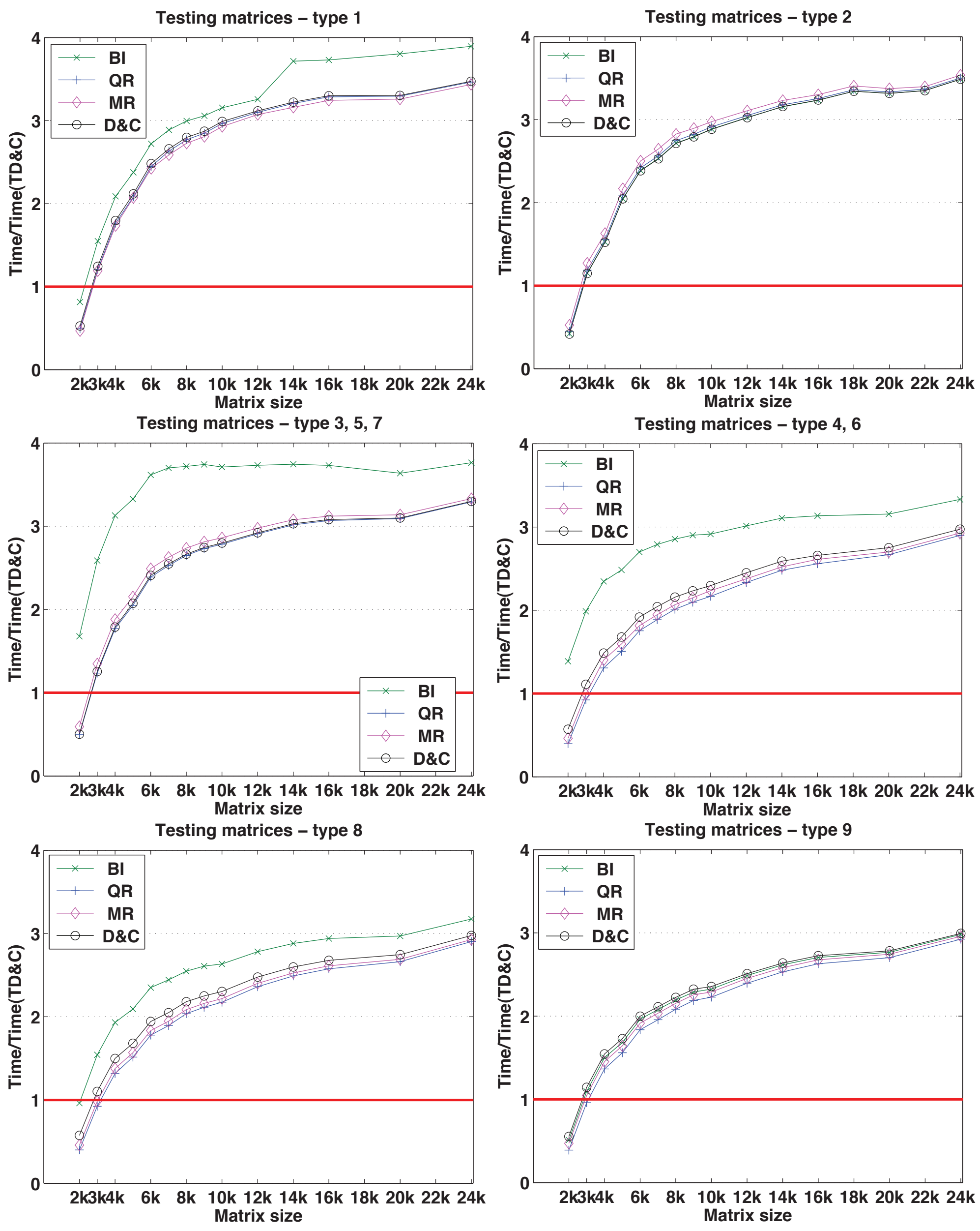


Fig. 6.4. *TD&C speedup as compared to BI, QR, MR, and D&C for all synthetic testing matrices from Table* 6.1. *For BI, QR, MR, and D&C, the MKL LAPACK-implementation routine DSYTRD is used to transform the dense symmetric matrix A to tridiagonal.*

to the fact that the DSYRDD routine is slightly faster than the DSYTRD routine. Thus, we show in Figure 6.5 only two representative graphs.

The results show that the TD&C symmetric eigenvalue solver algorithm performs better than all other represented algorithms and for all synthetic testing matrices for $n \geq 3000$. The results also indicate that the TD&C speedup ranges from two times faster for matrices of size $3000 \leq n \leq 10000$ to more than three times faster for

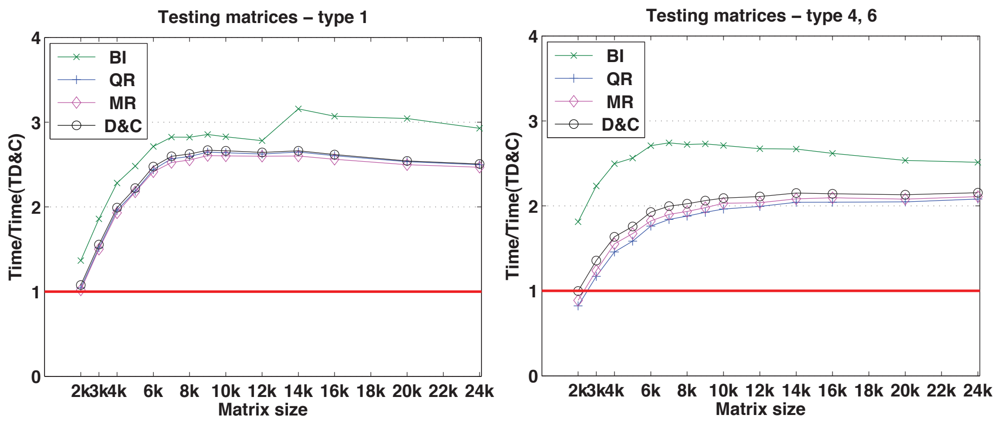


FIG. 6.5. *TD&C speedup as compared to BI, QR, MR, and D&C for all synthetic testing matrices from Table* 6.1. *For BI, QR, MR, and D&C, the MKL SBR-implementation routine DSYRDD is used to transform the dense symmetric matrix A to tridiagonal.*

matrices of size $n \geq 10000$. For small matrix sizes with $n \leq 3000$, TD&C runs slower than the other eigenvalue solvers. It is clear, from the algorithmic complexity study in section 5 that our TD&C algorithm requires $4/3n^3 + 6b^2n^2$ flops, which, for example, for $b = 20$ and $n = 3000$ means $4/3n^3 + 2400n^2 \approxeq 2n^3$ flops, while the other algorithms roughly perform $4/3n^3$ flops.

On the other hand, BI seems to be the slowest algorithm for computing eigenvalues, although its efficiency mostly depends on the distribution of the eigenvalues. Indeed, for matrices with strongly clustered eigenvalues (e.g., type 1), BI performance is slower than the other symmetric eigenvalue solver methods. Furthermore, the TD&C symmetric eigenvalue solver is able to take full advantage of a situation where a significant amount of deflations occur for special matrices of types 1 and 2. By avoiding unnecessary calculations, especially in the amalgamation phase thanks to deflations, the TD&C runs at even higher performance. This is also emphasized when comparing matrices from types 2 and 6. While a type 6 matrix represents the worst-case scenario for the TD&C symmetric eigenvalue solver in which less than 2% of deflations happen, a type 2 matrix corresponds to a matrix where a considerable amount of deflation occurs.

The TD&C performs considerably fewer flops in the latter case, and thus it runs more than twice as fast as when no deflations take place. For example, for a matrix of size $n = 16000$, the reduction to band form requires 145 seconds. The BD&C phase with lots of deflations requires 30 seconds instead of 74 seconds when no deflations occur, giving a total execution time of 175 seconds and 219 seconds, respectively (1.3-fold overall speedup). From a global point of view, the elapsed time and the performance results of the other methods (BI, QR, MR, and D&C) are very similar to one another. In addition to that, in the presence of deflation, the improvement is easily noticed for the TD&C algorithm, but it is rather moderate for the standard D&C algorithm. Therefore, once again, it is important to mention that when only eigenvalues are computed, BI, QR, MR, and D&C count only for $\mathcal{O}(n^2)$ on the overall algorithm complexity, and the leading term corresponding to the tridiagonal reduction phase is about $4/3n^3$. The time spent on this reduction phase can be as high as 90% of the total execution time when only eigenvalues are needed but more than 50% when eigenvectors are to be computed. Thus, no matter how D&C or the other methods

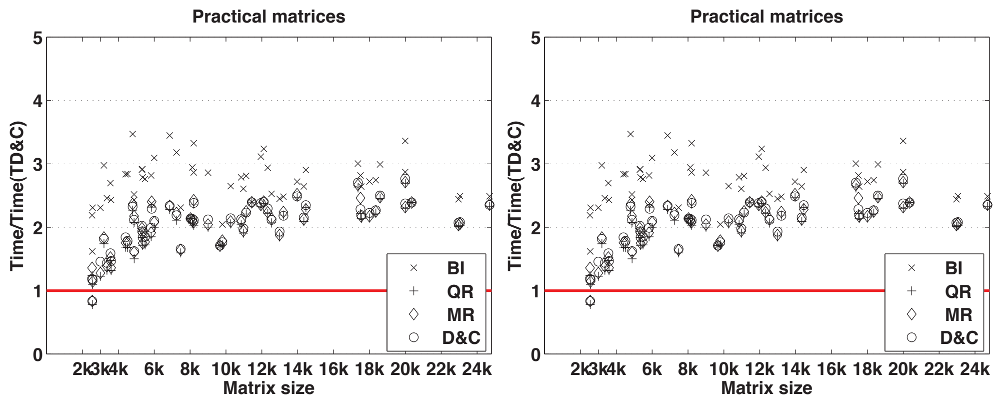


FIG. 6.6. *Timings comparison for practical matrices. The tridiagonalization has been done either by the MKL-LAPACK-DSYTRD (left) or by the MKL-SBR-DSYRDD (right).*

are improved, the impact on the overall algorithm performance is rather negligible for large matrix sizes.

6.7.3. Practical matrices (Table 6.3). The most important testing matrices are perhaps those which come from real applications. Figure 6.6 shows the speedup obtained by the TD&C symmetric eigenvalue solver as compared to BI, QR, MR, and D&C for a set of more than 100 matrices arising in different scientific and engineering areas. Similar to synthetic testing matrices, the TD&C symmetric eigenvalue solver algorithm outperforms all the other symmetric eigenvalue solvers, between two to three times faster, and achieves up to 36 Gflop/s.

6.8. Accuracy analysis.
This section is dedicated to the analysis of the TD&C symmetric eigenvalue solver accuracy as compared to the other four symmetric eigenvalue solvers: QR, BI, MR and D&C.

6.8.1. Metric definitions. For a given symmetric matrix B, computed eigenvectors $Q = [q_1, q_2, \dots, q_n]$ and their corresponding eigenvalues $\Lambda = \mathrm{diag}(\lambda_1, \lambda_2, \dots, \lambda_n)$, we use the following accuracy tests by using the LAPACK testing routines (DLANSY, DSTT21 for tridiagonal, and DSYT21 for dense symmetric):

$$\frac{\|I - QQ^T\|}{n \times ulp}, \tag{6.1}$$

which measures the orthogonality of the computed eigenvectors,

$$\frac{\|B - Q\Lambda Q^T\|}{\|B\| n \times ulp}, \tag{6.2}$$

which measures the accuracy of the computed eigenpairs, and

$$\frac{\|\lambda_i - \delta_i\|}{\|\lambda_i\| \times ulp}, \tag{6.3}$$

which measures the accuracy of the computed eigenvalues compared to the reference eigenvalue δ, which are the exact eigenvalues (analytically known), or the ones computed by the QR iteration routine using LAPACK (DSTEV). The value ulp represents the machine precision computed by the LAPACK subroutine DLAMCH. Its value on the Intel Xeon, where the accuracy experiments were conducted, is equal to 2.220446049259313E-16.

For most of the tests presented here, the original matrix is either generated from a multiplication of a given diagonal eigenvalue matrix by an orthogonal matrix or given directly as a symmetric matrix. Then, for the standard algorithms (QR, BI, MR, and D&C), the symmetric dense matrix is first reduced to tridiagonal form using the DSYTRD routine and then solved using one of the algorithms cited above. However, for our TD&C symmetric eigenvalue solver, the original matrix is first reduced to band form, and then solved using the band D&C method according to the approach described in this paper. Therefore, the matrix B considered in the metrics (6.1), (6.2), (6.3) is either tridiagonal obtained from the DSYTRD routine or in symmetric band form obtained from our TD&C symmetric eigenvalue solver.

6.8.2. Accuracy evaluations. We present the results obtained using the accuracy metrics defined above, for all sets of the matrices described in subsection 6.6. Our TD&C symmetric eigenvalue solver provides the three metrics with the same order of magnitude as compared to the other eigenvalue solvers.

Figure 6.7 and Figure 6.8 depict the losses of orthogonality (6.1) and the maximal residual norm (6.2) for practical and interesting matrix categories and synthetic testing matrix type, respectively. These figures allow for the study of the errors with respect to the matrix size n. The error of our TD&C symmetric eigenvalue solver de-

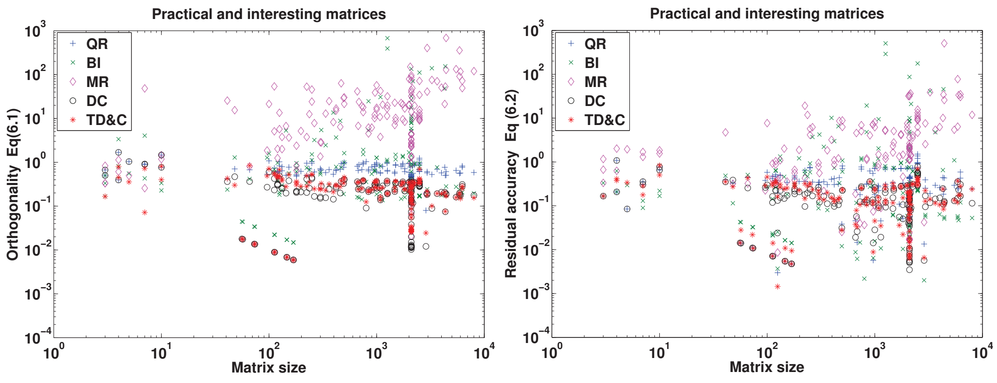


FIG. 6.7. *Summary of accuracy results observed on matrices from Tables* 6.2 *and* 6.3.

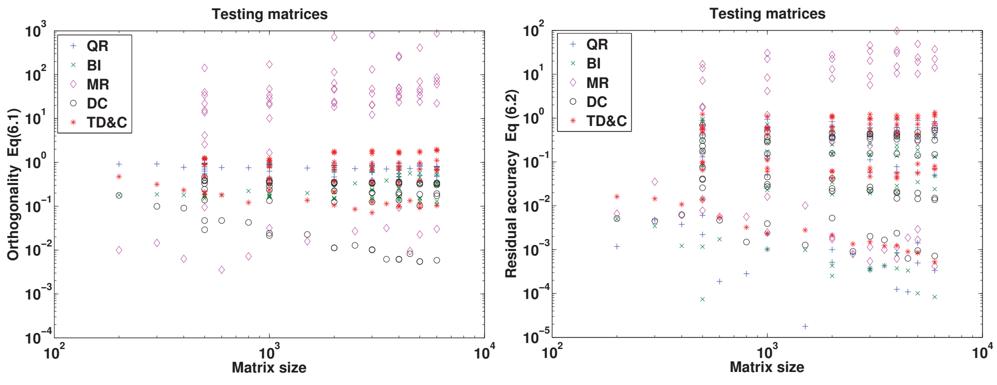


FIG. 6.8. *Summary of accuracy results observed on synthetic testing matrices from Table* 6.1.

creases when n increases, similarly to QR and D&C eigenvalue solvers. The observed accuracy is on the order of $\mathcal{O}(\sqrt{n} \cdot \varepsilon)$. However, it is clear that MRRR and BI do not achieve the same level of accuracy as QR, D&C, and TD&C. Finally, after measuring the eigenvector orthogonality (6.1), the residual norms (6.2) and the eigenvalue accuracy (6.3), our TD&C framework is one of the most accurate symmetric eigenvalue solver algorithms, which gives us a certain confidence on the quality of the overall TD&C symmetric eigenvalue solver presented in this paper.

7. Summary and future work.

The tile divide and conquer symmetric eigenvalue solver (TD&C) presented in this paper shows very promising results in terms of performance on multicore architecture, as well as in terms of numerical accuracy. TD&C has been extensively tested using different matrix types against other well-known symmetric eigenvalue solvers such as the QR iteration (QR), the bisection algorithm (BI), the standard divide and conquer (D&C), and the multiple relatively robust representations (MRRR). The performance results obtained for large matrix sizes and certain matrix types are very encouraging. The proposed TD&C symmetric eigenvalue solver reaches up to a 14-fold speed up compared to the state-of-the-art numerical open source library LAPACK V3.2, and up to a 4-fold speedup against the commercial numerical library Intel MKL V10.2. Our TD&C symmetric eigenvalue solver also proves to be one of the most accurate symmetric eigenvalue solvers available. The authors plan to eventually integrate this symmetric eigenvalue solver within the PLASMA library [34]. The authors are also currently looking at the eigenvector calculations. Indeed, the TD&C symmetric eigenvalue solver is not really appropriate for that purpose due to the large amount of extra flops required to accumulate them. The authors are investigating the possibility of applying a single rank-b modification (instead of b rank-one modifications), which would remove the cost of accumulating the subsequent orthogonal matrices to generate the eigenvectors.

Acknowledgment. The authors would like to thank the two anonymous reviewers for their insightful comments, which greatly helped to improve the quality of this article.

REFERENCES

[1] E. Agullo, B. Hadri, H. Ltaief, and J. Dongarra, *Comparative study of one-sided factorizations with multiple software packages on multi-core hardware*, in Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09), 2009, pp. 1–12.

[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 Users' Guide*, 3rd ed., SIAM, Philadelphia, PA, 1999.

[3] P. Arbenz, *Divide and conquer algorithms for the band symmetric eigenvalue problem*, Parallel Comput., 18 (1992), pp. 1105–1128.

[4] P. Arbenz, W. Gander, and G. H. Golub, *Restricted rank modification of the symmetric eigenvalue problem: Theoretical considerations*, Linear Algebra Appl., 104 (1988), pp. 75–95.

[5] P. Arbenz and G. H. Golub, *On the spectral decomposition of Hermitian matrices modified by low rank perturbations with applications*, SIAM J. Matrix Anal. Appl., 9 (1988), pp. 40–58.

[6] Y. Bai and R. C. Ward, *A parallel symmetric block-tridiagonal divide-and-conquer algorithm*, ACM Trans. Math. Softw., 33 (2007), 25.

[7] C. H. Bischof, B. Lang, and X. Sun, *Algorithm* 807: *The SBR toolbox—software for successive band reduction*, ACM Trans. Math. Softw., 26 (2000), pp. 602–616.

[8] J. R. Bunch, C. P. Nielsen, and D. C. Sorensen, *Rank-one modification of the symmetric eigenproblem*, Numer. Math., 31 (1978), pp. 31–48.

[9] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Comput., 35 (2009), pp. 38–53.

[10] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*, Parallel Comput. Syst. Appl., 35 (2009), pp. 38–53.

[11] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Orti, G. Quintana-Orti, and R. van de Geijn, *Supermatrix: A multithreaded runtime scheduling system for algorithms-by-blocks*, in Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'2008), Salt Lake City, UT, 2008, pp. 123–132.

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

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

[14] J. W. Demmel, O. A. Marques, B. N. Parlett, and C. Vömel, *Performance and Accuracy of LAPACK's Symmetric Tridiagonal Eigensolvers*, LAPACK Working Note 183, 2007.

[15] I. S. Dhillon. *A New $O(N(2))$ Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*, Ph.D. thesis, Technical Report UCB/CSD-97-971, University of California, Berkeley, CA, 1998.

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

[17] I. S. Dhillon, B. N. Parlett, and C. Vömel, *The design and implementation of the MRRR algorithm*, ACM Trans. Math. Softw., 32 (2006), pp. 533–560.

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

[19] W. N. Gansterer, J. Schneid, and C. Ueberhuber, *A low-complexity divide-and-conquer method for computing eigenvalues and eigenvectors of symmetric band matrices*, BIT, 41 (2001), pp. 967–976,

[20] W. N. Gansterer, R. C. Ward, and R. P. Muller, *An extension of the divide-and-conquer method for a class of symmetric block-tridiagonal eigenproblems*, ACM Trans. Math. Softw., 28 (2002), pp. 45–58.

[21] 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 (2003), pp. 65–85.

[22] K. Gates and P. Arbenz, *Parallel Divide and Conquer Algorithms for the Symmetric Tridiagonal Eigenproblem*, Technical Report 222, Department of Computer Science, ETH Zurich, 1994.

[23] G. H. Golub, *Some modified matrix eigenvalue problems*, SIAM Rev., 15 (1973), pp. 318–334.

[24] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed., Johns Hopkins University Press, Baltimore, MD, 1996.

[25] R. Grimes, H. Krakauer, J. Lewis, H. Simon, and S.-H. Wei, *The solution of large dense generalized eigenvalue problems on the Cray X-MP/24 with SSD*, J. Comput. Phys., 69 (1987), pp. 471–481.

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

[27] *Intel Math Kernel Library*, http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.

[28] I. C. F. Ipsen and E. R. Jessup, *Solving the symmetric tridiagonal eigenvalues problem on the hypercube*, SIAM J. Sci. Statist. Comput., 11 (1990), pp. 203–229.

[29] J. Kurzak, A. Buttari, and J. J. Dongarra, *Solving systems of linear equations on the CELL processor using Cholesky factorization*, IEEE Trans. Parallel Distrib. Syst., 19 (2008), pp. 1–11.

[30] J. Kurzak and J. Dongarra, *Fully Dynamic Scheduler for Numerical Scheduling on Multicore Processors*, LAPACK Working Note 220, UT-CS-09-643, Innovative Computing Lab, University of Tennessee, Knoxville, 2009.

[31] J. Kurzak and J. Dongarra, *QR factorization for the cell broadband engine*, Sci. Programming, 17 (2009), pp. 31–42.

[32] O. A. Marques, C. Vömel, J. W. Demmel, and B. N. Parlett, *Algorithm* 880: *A testing infrastructure for symmetric tridiagonal eigensolvers*, ACM Trans. Math. Softw., 35 (2008), pp. 8:1–8:13.

[33] R. M. Martin, *Electronic Structure: Basic Theory and Practical Methods*, Cambridge University Press, Cambridge, UK, 2008.

[34] *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures*, Version 2.4.5, University of Tennessee Knoxville, 2011.

[35] E. S. Quintana-Ortí and R. A. van de Geijn, *Updating an LU factorization with pivoting*, ACM Trans. Math. Softw., 35 (2008), 11.

[36] G. Quintana-Ortí, E. S. Quintana-Ortí, E. Chan, R. A. van de Geijn, and F. G. Van Zee, *Scheduling of QR factorization algorithms on SMP and multi-core architectures*, in PDP'08: Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, IEEE Computer Society, 2008, pp. 301–310.

[37] N. Rösch, S. Krüger, V. Nasluzov, and A. Matveev, *ParaGauss: The density functional program ParaGauss for complex systems in chemistry*, in High Performance Computing in Science and Engineering, Springer, New York, 2005, pp. 285–296,

[38] J. Rutter, *A Serial Implementation of Cuppen's Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem*, LAPACK Working Note 69, UT-CS-94-225, Department of Computer Science, University of Tennessee, Knoxville, 1994.

[39] *SMP Superscalar (SMPSs) User's Manual*, Version 2.0, Barcelona Supercomputing Center, http://www.bsc.es/media/1002.pdf, 2008.

[40] D. C. Sorensen and P. T. P. Tang, *On the orthogonality of eigenvectors computed by divide-and-conquer techniques*, SIAM J. Numer. Anal., 28 (1991), pp. 1752–1775.

[41] R. C. Thompson, *The behavior of eigenvalues and singular values under perturbations of restricted rank*, Linear Algebra Appl., 13 (1976), pp. 69–78.

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

[43] L. N. Trefethen and D. Bau III, *Numerical Linear Algebra*, SIAM, Philadelphia, PA, 1997.

[44] F. G. Van Zee, E. Chan, and R. A. van de Geijn, *libflame*, in Encyclopedia of Parallel Computing, Springer, New York, 2011, pp. 1010–1014.

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